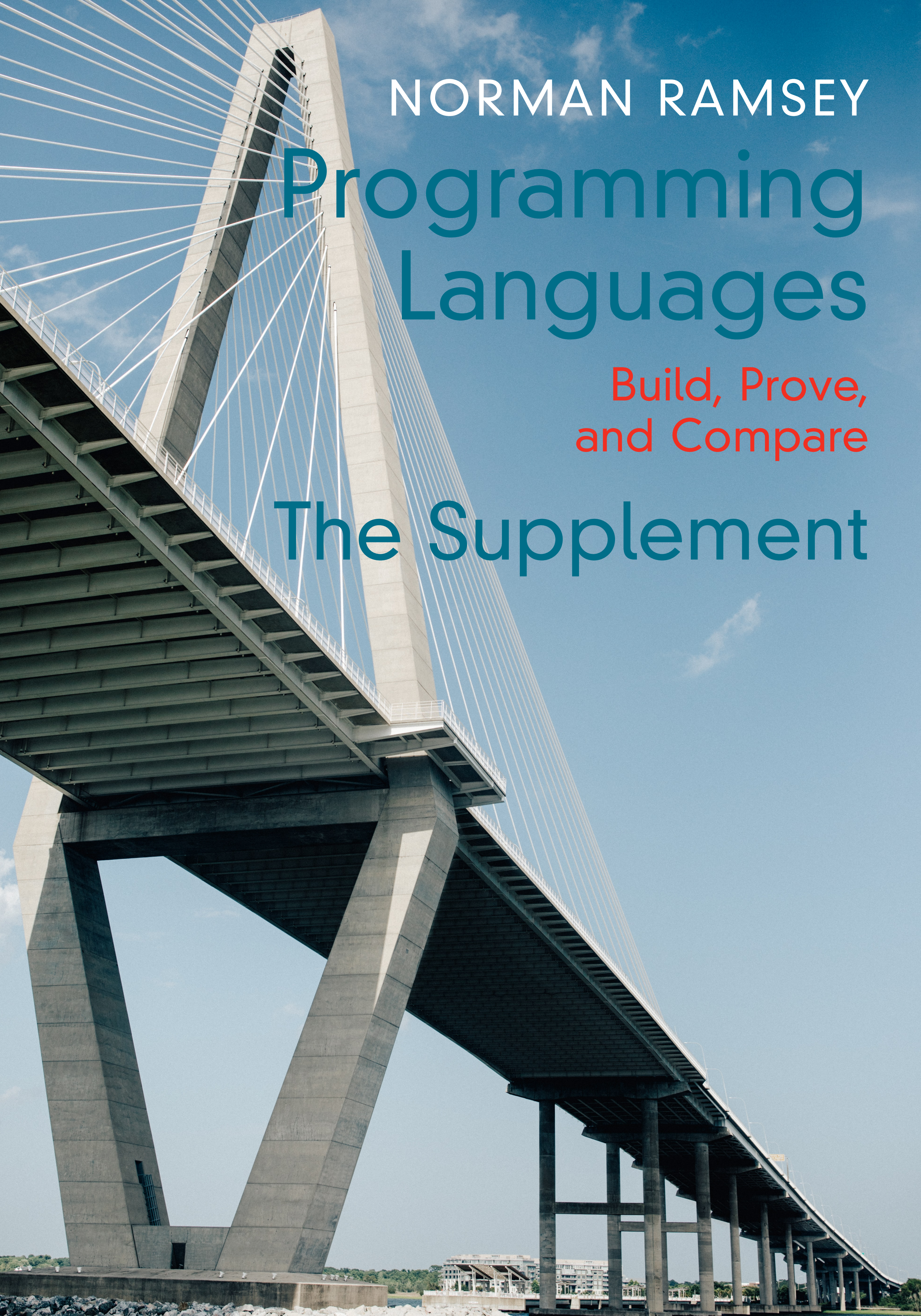


NORMAN RAMSEY

Programming Languages

Build, Prove, and Compare

The Supplement

Preface to the supplement

This volume is the Supplement to *Programming Languages: Build, Prove, and Compare*. It's not fully polished, and in places it's not even complete, but to complete and polish it would have held up publication, and nobody wanted that!

Supplement Contents

Supplemental topics S7

Long programming examples S115

Interesting infrastructure S159

Part III. Supplemental topics

A EBNF

Context-free grammars are a method of describing the syntax of programming languages. Context-free grammars are most often written in Backus-Naur Form, or BNF, in honor of the work done by John Backus and Peter Naur in creating the Algol 60 report. This book uses extended BNF, or simply EBNF, which makes it easier to specify optional and repeated items (Wirth 1977).

An EBNF grammar is a list of grammar *rules*. Each rule has the form:

$$A ::= \alpha$$

where A is a *nonterminal symbol*, and α is a collection of alternatives separated by vertical bars. Each alternative is a sequence, and in the simple case, each element of the sequence is either a nonterminal symbol or *literal text* (in `typewriter font`).

A nonterminal symbol represents all the phrases in a syntactic category. Thus, *def* represents all legal definitions, *exp* all legal expressions, *variable-name* all legal variable names, and so on. Literal text, on the other hand, represents characters that appear *as is* in syntactic phrases.

Consider the rule for *def* in Impcore:

$$\begin{aligned} \textit{def} ::= {} & \texttt{(val}\ \textit{variable-name exp}\texttt{)} \\ \mid {} & \textit{exp} \\ \mid {} & \texttt{(define}\ \textit{function-name}\ \texttt{(}\textit{formals}\texttt{)}\ \textit{exp}\texttt{)} \\ \mid {} & \texttt{(use}\ \textit{file-name}\texttt{)} \\ \mid {} & \textit{unit-test} \end{aligned}$$

This rule can be read as asserting that a legal *def* input is exactly one of the following:

- A left parenthesis followed by the word `val`, then a variable name, then a legal *exp*, and then a right parenthesis
- A legal *exp*
- A left parenthesis followed by the word `define`, then a function name, a left parenthesis, whatever is permitted as *formals*, a right parenthesis, an *exp*, and finally a right parenthesis
- A left parenthesis followed by the word `use`, then a file name, and then a right parenthesis
- A *unit-test*, which I won't belabor further.

A set of such rules is called a *context-free grammar*. It describes how to form the phrases of each syntactic category, in one or more ways, by combining phrases of other categories and specific characters in a specified order.

As another example, the phrase

```
(set x 10)
```

is a *def* input by the following reasoning:

- A *def* can be an *exp*.
- An *exp* can be a left parenthesis and the word `set`, followed by a *variable-name* and an *exp*, followed by a right parenthesis.
- A *variable-name* is a name, and a name is a sequence of characters which may be the sequence “`x`.”
- An *exp* can be a *literal*, a *literal* is a *numeral*, and `10` is a *numeral*.

The explanation above is not the whole story. In addition to a nonterminal symbol or literal text, any part of a rule’s right-hand side may contain a collection of alternatives that appear in brackets and are separated from one another by a “$|$” symbol (vertical bar). In EBNF, the vertical bar always means “alternative,” and different species of brackets stand for different things:

- Parentheses $(\cdots)$ stand for a choice of exactly one of the bracketed alternatives.
- Square brackets $\left[\cdots\right]$ stand for a choice of either nothing (the empty sequence), or exactly one of the bracketed alternatives.
- Braces $\left\{\cdots\right\}$ stand for a sequence of zero or more items, each of which is one of the bracketed alternatives.

For example, this rule shows that *formals* stands for a sequence of zero or more variable names:

$$\mathit{formals} ::= \left\{\mathit{variable\text{-}name}\right\}$$

Similarly, the EBNF phrase “$(\mathit{function\text{-}name}\ \left\{\mathit{exp}\right\})$” stands for a function name followed by a sequence of zero or more argument expressions, all in parentheses.

The topic of context-free grammars is an important one in computer science. It should be covered in depth in almost any introductory theory or compiler-construction book. Good sources include those from Aho et al. (2007), Barrett et al. (1986), and Hopcroft and Ullman (1979).

Appendix B contents

Algorithms for arithmetic

B

In the 21st century, many programmers take numbers for granted. Computer-science students rarely get more than a week's worth of instruction in the properties of floating-point numbers, and many programmers are barely aware that machine integers have limited precision. Arbitrary-precision arithmetic on integers or rational numbers is provided by so many languages that nobody really needs to know how the tricks are done. But if you want to learn, this appendix, together with Exercises 49 and 50 in Chapter 9 and Exercises 37 and 38 in Chapter 10, will teach you. And if you do both sets of exercises, you'll see how abstract data types compare with objects: when inspecting representations of multiple arguments, abstract data types make the abstractions easier to code but less flexible in use.

In programming as in math, numbers start with integers. You may not think of `int` as an abstract type, but it is. It is, however, an unsatisfying abstraction. Values of type `int` aren't true integers; they are *machine integers*. Although machine integers get bigger as hardware gets bigger—a typical machine integer occupies a machine word or half a machine word—they are always limited in precision. A 32-bit or 64-bit integer is good for many purposes, but some computations need more precision; examples include some cryptographic computations as well as exact rational arithmetic. *Arbitrary-precision* integer arithmetic is limited only by the amount of memory available on a machine. It is supported in many languages, and in highly civilized languages like Scheme, Smalltalk, and Python, arbitrary precision is the default.

Arbitrary-precision arithmetic makes a fine case study in information hiding. The concepts and algorithms are explained below, and I encourage you to implement them using both abstract data types (Chapter 9) and objects (Chapter 10). The similarities and differences among implementations illuminate what abstract data types are good at and what objects are good at.

Arbitrary-precision arithmetic begins with natural numbers—the nonnegative integers. Basic arithmetic includes addition, subtraction, multiplication, and division. An interface for natural numbers, written in Molecule, is shown in Figure B.1 on page S14. This interface exposes a couple of subtleties:

- The difference of two natural numbers isn't always a natural number; for example, $19 - 83$ is not a natural number. If `-` is used to compute such a difference, it halts the program with a checked run-time error. If you want such a difference not to halt your program, you can use continuation-passing style (Section 2.10): calling `(cps-minus` n_1 n_2 k_s k_f`)` computes the difference $n_1 - n_2$, and when the difference is a natural number, `cps-minus` passes it to success continuation k_s. Otherwise, `cps-minus` calls failure continuation k_f without any arguments.

- For efficiency, quotient and remainder are computed together. (It's done this way even in hardware.) Storing quotient and remainder is the purpose of record type `QR.pair`.

S14a. ⟨*nat.mcl* S14a⟩≡

```
(module-type NATURAL
  (exports [abstype t]
           [of-int : (int -> t)]   ; creator
           [+ : (t t -> t)]        ; producer
           [- : (t t -> t)]        ; producer, fails if negative
           [* : (t t -> t)]        ; producer
           [module [QR : (exports-record-ops pair
                                            ([quotient : t]
                                             [remainder : int]))]]
           [sdiv    : (t int -> QR.pair)]  ; producer
           [compare : (t t -> Order.t)]    ; observer
           [decimal : (t -> (@m ArrayList Int).t)] ; observer
              ; decimal representation, most significant digit first
           [cps-minus : (t t (t -> unit) (-> unit) -> unit)]))
                                   ; subtraction, using continuations,
                                   ; works even if negative
```

Figure B.1: An abstraction of natural numbers

- Long division—that is, division of a natural number by another natural number—is beyond the scope of this book. Instead, the interface can divide a natural number only by a (positive) machine integer. This “short division” is implemented by function `sdiv`.

A natural number should be representable easily and efficiently as a sequence of *digits* in a given *base*. The algorithms for simple arithmetic, which you may have learned in primary school, work digit by digit. In everyday life, we use base $b = 10$, and we write the most significant digit x_n on the left. In hardware, our computers famously use base $b = 2$; the word “bit” is a contraction of “binary digit.” Regardless of base, a single digit x_i is an integer in the range $0 \le x_i < b$. In arbitrary-precision arithmetic, we pick as large a b as possible, subject to the constraint that every arithmetic operation on digits must be doable in a single machine operation.

As taught to schoolchildren, arithmetic algorithms use base $b = 10$, but the algorithms are independent of b, as should be your implementation. The algorithms do depend, however, on the way you choose to represent a sequence of digits. I discuss two possibilities:

- You can represent a sequence as a list of digits, which is either empty or is a digit followed by a sequence of digits. If X is a natural number, one of the following two equations holds:

$$X = 0$$
$$X = x_0 + X' \cdot b.$$

where x_0 is the least-significant digit and X' is a natural number. (It is possible to start with x_n instead of x_0, but starting with x_0, which is the “little-endian” representation, simplifies the representation and all the computations.) A suitable representation might use an algebraic data type (Chapters 8 and 9):

S14b. ⟨*representation of natural numbers as a list of digits* S14b⟩≡

```
(data t
   [ZERO : t]
   [DIGIT-PLUS-NAT-TIMES-b : (int t -> t)])
```

Notation: Multiplication, visible and invisible

Mathematicians and physicists often multiply quantities simply by placing one next to another; for example, in the famous equation $E = mc^2$, m and c^2 are multiplied. But in a textbook on programming languages, this notational convention will not do. First, it is better for multiplication to be visible than to be invisible. And second, when one name is placed next to another, it usually means function application—at least that's what it means in ML, Haskell, and the lambda calculus.

Among the conventional infix operators, * is more suited to code than to mathematics, and the $\times$ symbol is better reserved to denote a Cartesian product in a type system. In this book, on the rare occasions when we need to multiply numbers, I write an infix $\cdot$, so Einstein's famous equation would be written $E = m \cdot c^2$.

Another possibility is to use objects: a class `NatZero` with no instance variables, and a class `NatNonzero` with instance variables x_0 and X'.

A good invariant, no matter what the representation, is that for either `(DIGIT-PLUS-NAT-TIMES-b` x_0 X'`)` or `NatNonzero`, x_0 and X' are not both zero. The abstraction function is

$$\begin{aligned} \mathcal{A}(\texttt{ZERO}) &= 0 \\ \mathcal{A}(\texttt{(DIGIT-PLUS-NAT-TIMES-b}\ x_0\ X'\texttt{)}) &= x_0 + X' \cdot b. \end{aligned}$$

- Alternatively, you can represent a sequence as an array of digits, that is, $X = x_0, \ldots, x_n$. The abstraction function is

$$\mathcal{A}(X) = \sum_{i=0}^{n} x_i \cdot b^i.$$

In both representations, every digit x_i satisfies the invariant $0 \le x_i < b$.

The design trade-offs are as follows: Using the list representation, the algorithms are easy to code, but the representation requires roughly double the space of the array representation. Using the array representation, not all the algorithms are as easy to code, but the representation requires half the space of the list representation. Algorithms for both representations are shown in the rest of this section.

B.1 Addition

Adding two digits doesn't always produce a digit. For example, if $b = 10$, the sum $3 + 9$ is not a digit. To express the sum, we say that it *carries out* 1, which we write $3 + 9 = 2 + 1 \cdot 10^1$. The carried 1 is added to the sum of the next digits, at which time it is called a "carry in," as in this example:

$$\begin{array}{r} {\scriptstyle 1} \\ 7\,3 \\ +\,8\,9 \\ \hline 1\,6\,2 \end{array}$$

The small 1 over the 7 is the "carry out" from adding 3 and 9, and it is "carried in" to the sum of 7 and 8, producing 16.

Table B.2: Metavariables used to describe multiprecision arithmetic

b	Base of multiprecision arithmetic
X, Y	A natural number that is added, subtracted, subtracted from, multiplied, or divided by
x_0, y_0	Least-significant digit ($X \bmod b, Y \bmod b$)
x_i, y_i	Digit i of a natural number
X', Y'	Sequence of most-significant digits (X div b, Y div b)
Z	Sum, difference, or product
z_i	Digit i of Z
c_i	Carry in at position i
c_{i+1}	Carry out at position i (also carry in at position $i + 1$)
d	Divisor
Q	Quotient
q_0	Least-significant digit of quotient ($Q \bmod b$)
q_i	Digit i of quotient, $0 \le q_i < b$
Q'	Most-significant digits of quotients (Q div b)
r	Remainder, always $0 \le r < d$
r'_i	"Remainder in" at digit i, $0 \le r'_i < d$
r_i	"Remainder out" at digit i, $0 \le r_i < d$

To turn the example into an algorithm, we start with the list representation, and we consider how to add nonzero natural numbers $X = x_0 + X' \cdot b$ and $Y = y_0 + Y' \cdot b$. We first add the two least-significant digits $x_0 + y_0$, then add any resulting carry out to $X' + Y'$. To specify the algorithm precisely, we resort to algebra.

The sum of X and Y can be expressed as

$$X + Y = (x_0 + X' \cdot b) + (y_0 + Y' \cdot b) = (x_0 + y_0) + (X' + Y') \cdot b.$$

Because sum $x_0 + y_0$ might be too big to fit in a digit, this right-hand side does not immediately determine a valid representation of the sum. To get a valid representation, we calculate the least-significant digit z_0 of the sum and the carry out c_1:

$$\begin{aligned} z_0 &= (x_0 + y_0) \bmod b \\ c_1 &= (x_0 + y_0) \operatorname{div} b. \end{aligned}$$

Now $x_0 + y_0 = z_0 + c_1 \cdot b$, and we can rewrite the sum as

$$X + Y = z_0 + (X' + Y' + c_1) \cdot b.$$

This right-hand side *does* immediately determine a good representation: z_0 can be represented as a digit, and the sum $X' + Y' + c_1$ can be represented as a natural number. The right-hand side also suggests that the general form of addition should compute sums of the form $X + Y + c$. Such sums can be expressed using a three-argument "add with carry" function, $adc(X, Y, c)$. Function adc is specified by these equations:

$$\begin{aligned} adc(0, Y, c_0) &= Y + c_0 \\ adc(X, 0, c_0) &= X + c_0 \\ adc(x_0 + X' \cdot b, y_0 + Y' \cdot b, c_0) &= z_0 + (X' + Y' + c_1) \cdot b, \\ &\qquad \text{where } z_0 = (x_0 + y_0 + c_0) \bmod b \\ &\qquad\qquad\quad c_1 = (x_0 + y_0 + c_0) \operatorname{div} b. \end{aligned}$$

In the example shown above, where we add 73 and 89,

$$x_0 = 3 \qquad X' = 7 \qquad y_0 = 9 \qquad Y' = 8 \qquad c_0 = 0 \qquad z_0 = 2 \qquad c_1 = 1.$$

Given an X and a Y represented as lists, function adc is most easily implemented recursively, using `case` expressions to scrutinize the forms of X and Y. It needs an auxiliary function to compute $Y + c_0$ and $X + c_0$, the specification of which is left as Exercise 11 in Chapter 9.

When X and Y are represented as arrays, function adc is not as easy to implement. A better approach instead loops on an index i; at each iteration, the loop computes one digit z_i and one carry bit c_{i+1}:

$$\begin{aligned} z_i &= (x_i + y_i + c_i) \bmod b, \\ c_{i+1} &= (x_i + y_i + c_i) \operatorname{div} b. \end{aligned}$$

The initial carry in c_0 is zero.

If X has n digits and Y has m digits, we require

$$X + Y = Z = \sum_{i=0}^{\max(m,n)+1} z_i \cdot b^i.$$

The computations of z_i and c_{i+1} are motivated by observing

$$\begin{aligned} X + Y &= \left(\sum_{i=0}^{n} x_i \cdot b^i\right) + \left(\sum_{j=0}^{m} y_j \cdot b^j\right) \\ &= \sum_{i=0}^{\max(m,n)} x_i \cdot b^i + y_i \cdot b^i \\ &= \sum_{i=0}^{\max(m,n)} (x_i + y_i) \cdot b^i \end{aligned}$$

and

$$x_i + y_i + c_i = z_i + c_{i+1} \cdot b.$$

In the example shown above, where we add 73 and 89,

$$\begin{aligned} z_0 + c_1 \cdot b &= x_0 + y_0 + c_0, && \text{where } x_0 = 3, y_0 = 9, c_0 = 0, z_0 = 2, c_1 = 1 \\ z_1 + c_2 \cdot b &= x_1 + y_1 + c_1, && \text{where } x_1 = 7, y_1 = 8, c_1 = 1, z_1 = 6, c_2 = 1 \\ z_2 + c_3 \cdot b &= x_2 + y_2 + c_2, && \text{where } x_2 = 0, y_2 = 0, c_2 = 1, z_2 = 1, c_2 = 0. \end{aligned}$$

B.2 Subtraction

The algorithm for subtraction resembles the algorithm for addition, but the carry bit is called a "borrow," and it works a little differently. If $Z = X - Y$, then digit z_i is computed from the difference $x_i - y_i - c_i$, where c_i is a borrow bit. If this difference is negative, you must borrow b from a more significant digit, exploiting the identity

$$z_{i+1} \cdot b^{i+1} + z_i \cdot b^i = (z_{i+1} - 1) \cdot b^{i+1} + (z_i + b) \cdot b^i.$$

If no more significant digit is available to borrow from, the difference is negative and therefore is not representable as a natural number—and the subtraction function must transfer control to a failure continuation (or halt with a checked run-time error).

An algorithm that uses the array representation can loop on i, just as for addition, and it can keep track of the borrow bit c_i at each iteration. An algorithm that uses the list representation can use a recursive function sbb (subtract with borrow), which is specified by these equations for $sbb(X, Y, c) = X - Y - c$:

$$\begin{aligned}
sbb(X,0,0) &= X \\
sbb(X,0,1) &= X - 1 \\
sbb(0, y_0 + Y' \cdot b, c) &= 0, && \text{if } y_0 = 0 \text{ and } Y' = 0 \text{ and } c = 0 \\
sbb(0, y_0 + Y' \cdot b, c) &= \textbf{error}, && \text{if } y_0 \neq 0 \text{ or } Y' \neq 0 \text{ or } c \neq 0 \\
sbb(x_0 + X' \cdot b, y_0 + Y' \cdot b, c) &= \qquad x_0 - y_0 - c + sbb(X', Y', 0) \cdot b, \\
&\qquad\qquad\qquad\qquad \text{if } x_0 - y_0 - c \geq 0 \\
sbb(x_0 + X' \cdot b, y_0 + Y' \cdot b, c) &= b + x_0 - y_0 - c + sbb(X', Y', 1) \cdot b, \\
&\qquad\qquad\qquad\qquad \text{if } x_0 - y_0 - c < 0
\end{aligned}$$

The specification of an algorithm for computing $X - 1$ is left as Exercise 11 in Chapter 9.

B.3 Multiplication

To compute the product of two natural numbers X and Y, we compute the partial products of all the pairs of digits, then add the partial products. Here's an example:

$$\begin{array}{cccc}
 & & 7 & 3 \\
 & & 8 & 9 \\
\hline
 & & 2 & 7 \\
 & 2 & 4 & \\
{\scriptstyle 1} & 6 & 3 & \\
5 & 6 & & \\
\hline
6 & 4 & 9 & 7
\end{array}$$

As in the case of addition, the product of two digits $x_i \cdot y_i$ might not be representable as a digit, so we compute

$$\begin{aligned}
z_{hi} &= (x_i \cdot y_i) \operatorname{div} b \\
z_{lo} &= (x_i \cdot y_i) \bmod b \\
x_i \cdot y_i &= z_{lo} + z_{hi} \cdot b,
\end{aligned}$$

and both z_{hi} and z_{lo} are representable as digits.

To multiply two natural numbers represented as lists, we can use these equations:

$$\begin{aligned}
X \cdot 0 &= 0 \\
0 \cdot Y &= 0 \\
(x_0 + X' \cdot b) \cdot (y_0 + Y' \cdot b) &= z_{lo} + (z_{hi} + x_0 \cdot Y' + X' \cdot y_0) \cdot b + (X' \cdot Y') \cdot b^2, \\
&\quad \text{where } z_{hi} = (x_0 \cdot y_0) \operatorname{div} b \\
&\qquad\qquad\; z_{lo} = (x_0 \cdot y_0) \bmod b.
\end{aligned}$$

These equations unpack *both* numbers into their constituent parts, so they are suitable for a language like Molecule, in which the representations of both multiplicands can be inspected (Section 9.7, page 555). That last equation unpacks into these steps:

1. Turn each single digit z_{lo}, z_{hi}, x_0, or y_0 into a natural number, by forming $z_{lo} = z_{lo} + 0 \cdot b$, and so on.
2. Use recursive calls to multiply natural numbers $x_0 \cdot Y'$, $X' \cdot y_0$, and $X' \cdot Y'$.
3. Add natural numbers z_{hi}, $x_0 \cdot Y'$, and $X' \cdot y_0$ into an intermediate sum S, then multiply $S \cdot b$ by forming the natural number $0 + S \cdot b$.
4. Compute $(X' \cdot Y') \cdot b^2$ by forming the natural number $0 + (0 + (X' \cdot Y') \cdot b) \cdot b$.
5. Add the three natural-number terms of the right-hand side.

For a language like μSmalltalk, in which only the representation of the receiver can be inspected, we can use these equations:

$$0 \cdot Y = 0$$
$$(x_0 + X' \cdot b) \cdot Y = x_0 \cdot Y + (X' \cdot Y) \cdot b$$

The product of two natural numbers $X \cdot Y$ or $X' \cdot Y$ can be computed by sending the `*` message to receiver X or X'. But a product like $x_0 \cdot Y$ has to be computed by sending a special-purpose message, say `multiplyBySmallInteger:`, to Y.

To multiply two natural numbers represented as arrays, we compute

$$\begin{aligned} X \cdot Y &= \left(\sum_i x_i b^i\right) \cdot \left(\sum_j y_j b^j\right) \\ &= \sum_i \sum_j (x_i \cdot y_j) \cdot b^{i+j} \end{aligned}$$

Again, to satisfy the representation invariant, each partial product $(x_i \cdot y_j) \cdot b^{i+j}$ has to be split into two digits $((x_i \cdot y_j) \bmod b) \cdot b^{i+j} + ((x_i \cdot y_j) \operatorname{div} b) \cdot b^{i+j+1}$. Then all the partial products are added.

B.4 Short division

Long division, in which you divide one natural number by another, is beyond the scope of this book. Consult Hanson (1996) or Brinch Hansen (1994). But short division, in which you divide a big number by a digit, is within the scope of the book, and it is used to implement the `print` method. To convert a large integer to a sequence of decimal digits that can be printed, we divide the large integer by 10 to get its least significant digit (the remainder), then recursively convert the quotient.

Here is an example of short division in decimal. When 1528 is divided by 7, the result is 218, with remainder 2:

$$\begin{array}{r@{\,}l} & \;0\;\;\;2\;\;\;1\;\;\;8 \\ 7\,\big) & \overline{1\;{}^{1}5\;{}^{1}2\;{}^{5}8} \text{ remainder 2} \end{array}$$

Short division works from the most-significant digit of the dividend down to the least-significant digit:

1. We start off dividing 1 by 7, getting 0 with remainder 1. Quotient 0 goes above the line (producing the most-significant digit of the overall quotient), and the remainder is multiplied by 10 and added to the next digit of the dividend (5) to produce 15.

2. When 15 is divided by 7, quotient 2 goes above the line (producing the next digit of the overall quotient), and remainder 1 is combined with the next digit of the dividend (2) to produce 12.

3. When 12 is divided by 7, quotient 1 goes above the line (producing the next digit of the overall quotient), and remainder 5 is combined with the next digit of the dividend (8) to produce 58.

4. When 58 is divided by 7, quotient 8 goes above the line (producing the final digit of the overall quotient), and remainder 2 is the overall remainder.

To turn the example into an algorithm, we consider large-integer dividend X divided by small-integer divisor d, from which we compute large-integer quotient Q and small-integer remainder r, satisfying

$$X = Q \cdot d + r \qquad\qquad 0 \le r < d.$$

The algorithm is easiest to specify when X is represented as a list of digits.

If X is zero, both Q and r are also zero. If X is nonzero, then it has the form $x_0 + X' \cdot b$, and we start with the most-significant digits X'. We recursively divide X' by d, giving quotient Q' and remainder r'. To get the final quotient $Q = q_0 + Q' \cdot b$ and remainder r, we divide machine integer $x_0 + r' \cdot b$ by d:

$$\begin{aligned} X = x_0 + X' \cdot b = (q_0 + Q' \cdot b) \cdot d + r \\ \text{where } q_0 &= (x_0 + r' \cdot b) \operatorname{div} d \\ r &= (x_0 + r' \cdot b) \bmod d. \end{aligned}$$

In our example above,

$$\begin{array}{rcc} X = 1528 & d = 7 & q_0 = 8 \\ x_0 = 8 & Q' = 21 & Q = 218 \\ X' = 152 & r' = 5 & r = 2. \end{array}$$

When X is represented as an array, the algorithm loops *down* over index i, starting with $i = n$ and going down to $i = 0$. At each iteration, the algorithm computes a digit q_i of the quotient, and it computes an intermediate remainder r_i. That remainder is then named r'_{i-1}, where it is combined with digit x_{i-1} to be divided by d. Here are the equations:

$$\begin{aligned} q_i &= (r'_i \cdot b + x_i) \operatorname{div} d & \qquad r &= r_0 \\ r_i &= (r'_i \cdot b + x_i) \bmod d & r'_{i-1} &= r_i \\ & & r'_n &= 0. \end{aligned}$$

In the example on page S19,

$$\begin{array}{lll} x_3 = 1 & d = 7 & \begin{aligned} q_3 &= (0 \cdot 10 + 1) \operatorname{div} 7 &&= 0 \\ r_3 &= (0 \cdot 10 + 1) \bmod 7 &&= 1 \end{aligned} \\ x_2 = 5 & r'_3 = 0 & \begin{aligned} q_2 &= (1 \cdot 10 + 5) \operatorname{div} 7 &&= 2 \\ r_2 &= (1 \cdot 10 + 5) \bmod 7 &&= 1 \end{aligned} \\ x_1 = 2 & & \begin{aligned} q_1 &= (1 \cdot 10 + 2) \operatorname{div} 7 &&= 1 \\ r_1 &= (1 \cdot 10 + 2) \bmod 7 &&= 5 \end{aligned} \\ x_0 = 8 & & \begin{aligned} q_0 &= (5 \cdot 10 + 8) \operatorname{div} 7 &&= 8 \\ r_0 &= (5 \cdot 10 + 8) \bmod 7 &&= 2. \end{aligned} \end{array}$$

S21. ⟨*Molecule's predefined module types* S21⟩≡

```
(module-type INT
  (exports [abstype t]                [<  : (t t -> Bool.t)]
           [+ : (t t -> t)]           [<= : (t t -> Bool.t)]
           [- : (t t -> t)]           [>  : (t t -> Bool.t)]
           [* : (t t -> t)]           [>= : (t t -> Bool.t)]
           [/ : (t t -> t)]           [=  : (t t -> Bool.t)]
           [negated : (t -> t)]       [!= : (t t -> Bool.t)]
           [print   : (t -> Unit.t)]
           [println : (t -> Unit.t)]))
```

Figure B.3: An interface to integer arithmetic

§B.5
Choosing a base

S21

B.5 Choosing a base of natural numbers

The algorithms above are independent of the base b. This base should be hidden from client code, so you can choose any base that you want. What base should you choose? For best performance, choose the largest b such that every intermediate value of every computation can be represented as an atomic value.

Should you find yourself working with assembly code or with machine instructions, your atomic value would be a machine word. You would have access to a hardware "flag" or other register that could hold a carry bit or borrow bit, and also to an "extended multiply" instruction that would provide the full two-word product of two one-word multiplicands. The result of every intermediate computation would be right there in the hardware, and you would choose $b = 2^k$, where k would be the number of bits in a machine word.

When you're working with a high-level language, your atomic value is a value of type `int`. But you probably *don't* have access to an add-with-carry instruction or an extended-multiply instruction. More likely, you are stuck with an `int` that has only 32 or 64 bits—or in some cases, even fewer bits. You have to choose b small enough so that an `int` can represent any possible intermediate result:

- To implement addition and subtraction, you must be able to represent a sum which may be as large as $2 \cdot b - 1$.
- To implement multiplication, you must be able to represent a partial product which may be as large as $(b - 1)^2$.
- To implement division, you must be able to represent the combination of a remainder with a digit, which may be as large as $(d-1) \cdot b + (b-1)$. If $d \leq b$, this combination may be as large as $b^2 - 1$.

Depending on niceties of signed versus unsigned arithmetic, and whether values of type `int` occupy 32 bits or 64, you can usually get good results with $b = 2^{15}$ or $b = 2^{31}$. (Using a power of 2 makes computations $\bmod b$ and $\operatorname{div} b$ easy and fast.)

B.6 Signed-integer arithmetic

Arithmetic on natural numbers can be leveraged to implement arithmetic on full, signed integers. One possible interface, written in Molecule, is shown in Figure B.3. While machine arithmetic typically uses a two's-complement representation of integers, for arbitrary-precision arithmetic, I recommend a representation that tracks the *sign* and *magnitude* of an integer. If you're using Molecule, here are three good representations:

- Represent the magnitude and sign independently.

- Define an algebraic data type that encodes the sign in a value constructor, and apply the value constructor to the magnitude, as in `(NEGATIVE mag)`.

- Define an algebraic data type with *three* value constructors: one each for positive numbers, negative numbers, and zero. A value constructor for a positive or negative number is applied to a magnitude. The value constructor for zero is an integer all by itself.

If you're using μSmalltalk, there's only one sensible choice: as described in Section 10.8, use classes `LargePositiveInteger` and `LargeNegativeInteger`.

Sign and magnitude can also be used to specify the abstraction, and if you do so, you can specify most operations using algebraic laws. Some examples:

$$\begin{array}{rcl@{\qquad}rcl}
+N + +M &=& +(N + M) & +N < +M &=& N < M \\
+N + -M &=& +(N - M), \text{ when } N \geq M & +N < -M &=& \texttt{\#f} \\
+N + -M &=& -(M - N), \text{ when } N < M & \texttt{negated}(+N) &=& -N \\
+N + 0 &=& +N & \texttt{negated}(0) &=& 0
\end{array}$$

The implementation of these laws depends on the programming language. If you're using abstract data types in Molecule, your code can inspect the representations of two integers at once, and the signed-integer operations can be implemented by pattern matching on pairs. If you're using objects in μSmalltalk, your code will have to identify some representations using double dispatch (Section 10.8.3).

Appendix C contents

C Extensions to algebraic data types

As I wrap up this book, one of the most interesting frontiers in programming languages is the design of advanced type systems. People want type systems that do more, ideally without giving up type inference. And the algebraic data types described in Chapter 8 *can* do more. In this appendix I describe two extensions that are now well established.

The first extension is *existential quantification*. Existential quantification enables us to hide information about representation, which in turn enables us to create mixed representations that support an "open world." Existential quantification provides a nice type-theoretic model for object-oriented programming: an object's private representation is existentially quantified. As evidence, I present an implementation of *shapes*; you can compare the examples below with the examples in Chapter 10, which use objects.

The second extension is *generalized* algebraic data types, usually abbreviated to GADTs. GADTs help refine information about type variables. Normally, all we know about a type variable is that it stands for information about an unknown type. But by using GADTs, we can look at a value constructor and get additional information, limited in scope, about a type parameter to a datatype constructor.

To implement the first extension, existentials, requires minimal changes to type inference and no changes to constraint solving. The type theory appears below, and the code is in Appendix S. To implement the second extension, GADTs, requires too much change to my interpreter: a more general representation of types, many changes to type inference, and a much more sophisticated constraint solver. These changes are beyond the scope of this book.

C.1 Existentials

Existential types enable us to hide what is usually known. They provide a great model for object-oriented languages, in which what is hidden is the representation of an object. And like objects, existential types enable new ways of thinking about data structures and their evolution. I present a simple example on page S27 below, which you can compare with the opening example of Chapter 10. But before you look at the example, I had better show how existential types work.

Trivial example: transparent and opaque boxes

As you know, a value of algebraic data type is constructed by applying a value constructor to arguments. What do we know about the arguments? If we know the type of the result value, and we know what value constructor was applied, then we know everything there is to know about the types of the arguments. Formally, when we know τ, we know each τ_i:

$$\frac{\begin{array}{c}\Gamma \vdash K : \tau_1 \times \cdots \times \tau_m \to \tau \\ \Gamma, \Gamma'_i \vdash p_i : \tau_i, \quad 1 \le i \le m \\ \Gamma' = \Gamma'_1 \uplus \cdots \uplus \Gamma'_m\end{array}}{\Gamma, \Gamma' \vdash (K\ p_1\ \cdots\ p_m) : \tau}. \qquad (\textsc{PatVcon})$$

Existentials let us hide information about τ_i's.

Before we start hiding things, let's start with an ordinary algebraic data type in which nothing is hidden: a transparent box.

S26a. ⟨*existential transcript* S26a⟩≡ S26b ▷

```
-> (data (* => *) transparent-box
     [TBOX : (forall ['a] ('a -> (transparent-box 'a)))])
transparent-box :: (* => *)
TBOX : (forall ['a] ('a -> (transparent-box 'a)))
```

We can put a value in a box, then take it again, and we never lose track of its type:

S26b. ⟨*existential transcript* S26a⟩+≡ ◁ S26a S26c ▷

```
put-in   : (forall ['a] ('a -> (transparent-box 'a)))
take-out : (forall ['a] ((transparent-box 'a) -> 'a))
```

```
-> (val put-in TBOX)
-> (define take-out (box) (case box [(TBOX a) a]))
```

Transparent boxes are polymorphic; a transparent box can hold a value of any type we like.

S26c. ⟨*existential transcript* S26a⟩+≡ ◁ S26b S26d ▷

```
-> (val box1 (put-in 'answer))
(TBOX answer) : (transparent-box sym)
-> (val box2 (put-in 42))
(TBOX 42) : (transparent-box int)
```

But we can't make a *list* of `box1` and `box2`—they have different types:

S26d. ⟨*existential transcript* S26a⟩+≡ ◁ S26c S26e ▷

```
-> (list2 box1 box2)
type error: cannot make int equal to sym
```

If `box1` and `box2` could somehow hide the types of their contents, then we could put them on a list. To make an *opaque* box that hides the type of its contents, I use an existential:[1]

S26e. ⟨*existential transcript* S26a⟩+≡ ◁ S26d S26f ▷

```
-> (data * opaque-box
     [OBOX : (forall ['a] ('a -> opaque-box))])
opaque-box :: *
OBOX : (forall ['a] ('a -> opaque-box))
```

The opaque box *doesn't take a type parameter*. If I put something in an opaque box, its type is hidden:

S26f. ⟨*existential transcript* S26a⟩+≡ ◁ S26e S27a ▷

```
hide : (forall ['a] ('a -> opaque-box))
```

```
-> (val hide OBOX)
-> (val box3 (hide 'the-body))
(OBOX the-body) : opaque-box
-> (val box4 (hide (lambda (n) (+ (* 2 n) 1))))
(OBOX <function>) : opaque-box
-> (val hidden-answer (hide 42))
(OBOX 42) : opaque-box
```

[1] Please tolerate, for the moment, the lunacy of calling something "existential" when it is written `forall`.

And once something is hidden, there's no way to reveal it. The definition of `reveal` here is exactly the same as the definition of `take-out` above, except it uses value constructor `OBOX` instead of `TBOX`:

S27a. ⟨*existential transcript* S26a⟩+≡ ◁ S26f S27b ▷

```
-> (define reveal (box) (case box [(OBOX a) a]))
type error: in choice [(OBOX a) a], right-hand side has type skolem type 23, ...
```

The error message complains that "skolem type 23" is an "escaping skolem type." The *skolem type* (page S31, named for Norwegian mathematician Thoralf Skolem) is a proxy for the unknown type of the value inside the box. Even if we know, as programmers, what the value is, the type system won't let us compute with it. For example, even though I know the result of applying the function in `box4` should be an integer—there are no mysterious "escaping" skolem types—the type system won't let me do it.

S27b. ⟨*existential transcript* S26a⟩+≡ ◁ S27a S27c ▷

```
-> (case box4 [(OBOX f) (f 7)])
type error: cannot make skolem type 24 equal to (int -> 'a)
```

The type system will not let me know that `f` is a function. It will, however, let me make a list of opaque boxes whose contents have different types:

S27c. ⟨*existential transcript* S26a⟩+≡ ◁ S27b S27d ▷

```
-> (list2 box3 box4)
((OBOX the-body) (OBOX <function>)) : (list opaque-box)
```

Because you can't do anything with the contents, the opaque box is useless. But it illustrates the mechanism, which I now deploy in a more compelling example.

Using existentials to create an open-world representation: shapes

Here I use existentials to develop a library for creating two-dimensional images from *shapes*. The library is based on ideas from object-oriented programming, in which the *representation* of each shape is private, but the *operations* available to perform on shapes are public (Chapter 10). I begin by using algebraic data types, in the standard way, to define an abstraction with multiple representations: I define a type with one value constructor per representation.

S27d. ⟨*existential transcript* S26a⟩+≡ ◁ S27c S27e ▷

```
-> (record pt ([x : int] [y : int]))  ;; a point on the plane
-> (implicit-data closed-shape
       [CIRCLE    of pt int]  ;; center and radius
       [RECTANGLE of pt pt])  ;; lower-left and upper-right corners
```

The type is called `closed-shape` because it embodies a *closed-world assumption*: once the type is defined, no new shapes can be added.

I want to implement three operations on shapes: scale a shape, translate a shape, and draw a shape. To scale something, I define a multiplier that says by how many thousandths the size of a shape should be multiplied.

S27e. ⟨*existential transcript* S26a⟩+≡ ◁ S27d S27f ▷

I start by scaling points and integers.

S27f. ⟨*existential transcript* S26a⟩+≡ ◁ S27e S28a ▷

```
-> (define scale-int (thousandths n)
     (/ (+ (* thousandths n) 500) 1000))
-> (define scale-pt (mult p)
     (make-pt (scale-int mult (pt-x p)) (scale-int mult (pt-y p))))
```

```
scale-int : (int int -> int)
scale-pt  : (int pt  -> pt)
```

Now I can scale shapes by doing a case analysis.

S28a. ⟨*existential transcript* S26a⟩+≡ ◁S27f S28b▷

```
                             scale-closed-shape : (int closed-shape -> closed-shape)
-> (define scale-closed-shape (f shape)
      (case shape
        [(CIRCLE center radius) (CIRCLE (scale-pt f center) (scale-int f radius))]
        [(RECTANGLE ll ur)      (RECTANGLE (scale-pt f ll) (scale-pt f ur))]))
```

I can implement translation and drawing in the same way. But the library isn't very useful, because it can't be extended with new shapes. What if I want an ellipse? Or a line? Or an arrow? Or a triangle? Or a list of shapes, one atop the next? Not one of these shapes can be represented using `closed-shape`. If you're limited to plain, ordinary algebraic data types, there's not much you can do. The usual technique is:

1. Extend the definition of `closed-shape` with new value constructors.
2. Extend the `scale-closed-shape` function with new cases.
3. Extend the `translate-closed-shape` function with new cases.
4. Extend the `draw-closed-shape` function with new cases.

Not only is this technique tedious, but if every program that uses shapes has to change the source code, there is no way to put the code into a library that many programs can share.

The damage can be mitigated by using type parameters and higher-order functions, but there is a better way: suppose we use existentials to hide the exact representations of shapes, and instead focus on the three operations of scaling, translation, and drawing. If we have those operations, for any shape, we can put them into a record, which is a central idea of object-oriented programming:

S28b. ⟨*existential transcript* S26a⟩+≡ ◁S28a S28c▷

```
                 make-shapely :
                 (forall ['a] ((int 'a -> 'a) (pt 'a -> 'a) ('a -> unit) -> (shapely 'a)))
                 shapely-scale     : (forall ['a] ((shapely 'a) -> (int 'a -> 'a)))
                 shapely-translate : (forall ['a] ((shapely 'a) -> (pt 'a -> 'a)))
                 shapely-draw      : (forall ['a] ((shapely 'a) -> ('a -> unit)))
-> (record ('a) shapely
     ([scale     : (int 'a -> 'a)]
      [translate : (pt 'a -> 'a)]
      [draw      : ('a -> unit)]))
```

Now we can represent a shape as an opaque package containing a representation of type β—I'm not going to let you see what it is—and a record of operations of type `(shapely` β`)`.

S28c. ⟨*existential transcript* S26a⟩+≡ ◁S28b S28d▷

```
-> (data * shape
      [SHAPE : (forall ['b] ('b (shapely 'b) -> shape))])   ;; existential 'b
shape :: *
SHAPE : (forall ['b] ('b (shapely 'b) -> shape))
```

Here's how we can scale a shape without knowing its representation:

S28d. ⟨*existential transcript* S26a⟩+≡ ◁S28c S29a▷

```
-> (define scale-shape (mult s)                 scale-shape : (int shape -> shape)
     (case s
       [(SHAPE b operations)
            (SHAPE ((shapely-scale operations) mult b) operations)]))
scale-shape : (int shape -> shape)
```

And translate:

S29a. ⟨*existential transcript* S26a⟩+≡ ◁S28d S29b▷

`translate-shape : (pt shape -> shape)`

```
-> (define translate-shape (vector s)
     (case s
       [(SHAPE b operations)
            (SHAPE ((shapely-translate operations) vector b) operations)]))
-> (define translate-pt (vector pt)
     (case (PAIR vector pt)
       [(PAIR (make-pt x1 y1) (make-pt x2 y2))
        (make-pt (+ x1 x2) (+ y1 y2))]))
```

And draw:

S29b. ⟨*existential transcript* S26a⟩+≡ ◁S29a S29c▷

`draw-shape : (shape -> unit)`

```
-> (define draw-shape (s)
     (case s
       [(SHAPE b operations)
            ((shapely-draw operations) b)]))
```

Now if we had a shape, we would know what to do with it. How do we make a shape? Choose a representation, and supply the relevant operations. Here's a circle:

S29c. ⟨*existential transcript* S26a⟩+≡ ◁S29b S29d▷

`circle : (pt int -> shape)`

```
-> (implicit-data circle [C of pt int])
                         ; (C center radius)
-> (use postscript.uml) ;; load PostScript drawing library
-> (val circle-ops
        (make-shapely
            (lambda (mult c)
                (case c [(C center radius)
                            (C (scale-pt mult center) (scale-int mult radius))]))
            (lambda (vec c)
                (case c [(C center radius) (C (translate-pt vec center) radius)]))
            (lambda (c)
                (case c [(C (make-pt x y) r) (ps-draw-circle x y r)]))))
-> (define circle (center radius)
     (SHAPE (C center radius) circle-ops))
```

I can make a disk using the same representation, changing only the drawing function.

S29d. ⟨*existential transcript* S26a⟩+≡ ◁S29c S29e▷

`disk : (pt int -> shape)`

```
-> (val disk
     (let* ([draw (lambda (c)
                    (case c ((C (make-pt x y) r) (ps-draw-disk x y r))))])
       (case circle-ops
         [(make-shapely scale translate _)
             (lambda (center radius)
                (SHAPE (C center radius) (make-shapely scale translate draw)))])))
```

Here is a line, which I represent as a list containing two points. I build the operator record, then return a function that makes shapes using that record.

S29e. ⟨*existential transcript* S26a⟩+≡ ◁S29d S30a▷

`line : (pt pt -> shape)`

```
-> (val line
     (let* ([scale (lambda (mult pts) (map ((curry scale-pt)     mult) pts))]
            [trans (lambda (vec  pts) (map ((curry translate-pt) vec)  pts))]
            [draw  (lambda (pts) (ps-draw-polyline '1.5 pt-x pt-y pts))]
            [ops   (make-shapely scale trans draw)])
       (lambda (p1 p2) (SHAPE (list2 p1 p2) ops))))
```

As my final shape, I define a list of shapes, drawn in order, to be a shape. Again I build the record and return a function.

S30a. ⟨*existential transcript* S26a⟩+≡ ◁S29e S30b▷

`shapes : ((list shape) -> shape)`

```
-> (val shapes
     (let* ([scale (lambda (mult shapes) (map ((curry scale-shape)     mult) shapes))]
            [trans (lambda (vec  shapes) (map ((curry translate-shape) vec)  shapes))]
            [draw  ((curry app) draw-shape)]
            [ops   (make-shapely scale trans draw)])
       (lambda (shapes) (SHAPE shapes ops))))
```

Now I can define a target shape:

S30b. ⟨*existential transcript* S26a⟩+≡ ◁S30a S30c▷

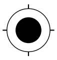
`target : shape`

```
-> (val target
      (let* ([origin (make-pt 0 0)]
             [center (disk origin 9)]
             [ring   (circle origin 15)]
             [tick   (lambda (x1 x2 y1 y2) (line (make-pt x1 x2) (make-pt y1 y2)))]
             [tick1  (tick  15   0  18   0)]
             [tick2  (tick -15   0 -18   0)]
             [tick3  (tick   0  15   0  18)]
             [tick4  (tick   0 -15   0 -18)])
        (shapes (list6 center ring tick1 tick2 tick3 tick4))))
```

And convert it to a PostScript file:

S30c. ⟨*existential transcript* S26a⟩+≡ ◁S30b

```
-> (define psfile (shape)
     (begin (println '%!PS-Adobe-1.0)
            (draw-shape shape)))
-> (psfile (translate-shape (make-pt 300 600) (scale-shape 2000 target)))
%!PS-Adobe-1.0
300 600 18 0 360 arc closepath 0.0 setgray fill
300 600 30 0 360 arc closepath stroke
1.5 setlinewidth newpath 330 600 moveto 336 600 lineto 0.0 setgray stroke
1.5 setlinewidth newpath 270 600 moveto 264 600 lineto 0.0 setgray stroke
1.5 setlinewidth newpath 300 630 moveto 300 636 lineto 0.0 setgray stroke
1.5 setlinewidth newpath 300 570 moveto 300 564 lineto 0.0 setgray stroke
UNIT : unit
```

If the output is placed in a file `target.ps`, most document viewers can display this glyph: 

Explanation and theory of existentials

To understand how existential types work and how they are implemented, let's try to build intuition by relating types to logical formulas. A logical formula $\forall x.P$ says that proposition P holds for *any* value of x—you can choose any x you like. But the logical formula $\exists x.P$ says that proposition P holds for *one particular* value of x—you don't get to choose x. In the existential formula, somebody else has chosen the value of x, and you don't know what value they've chosen.

Types work the same way. The type $\forall\alpha.\tau$ is a quantified type that can be instantiated by choosing *any* type τ' that you like, and substituting τ' for α in τ. The type $\exists\alpha.\tau$ is a quantified type that *can't* be instantiated any way you like. Somebody else has already chosen a τ', and the type you have access to is τ with the unknown τ' substituted for α.

Existential types have many honorable uses in programming languages, usually to formalize language constructs that hide information. But the use of existential types to describe value constructors is a bit startling: the type of a value constructor can be *either* universally quantified *or* existentially quantified, depending on the context in which it occurs. This context-dependent typing can be understood most easily in a very simple example: the opaque box (page S26). When it's used as a *value,* the value constructor `OBOX` has type $\forall\alpha.\alpha \to$ `opaque-box`. That is, you can choose a value of any type you like and put it in the box. But when it's used as a *pattern,* the value constructor `OBOX` has type $(\exists\alpha.\alpha) \to$ `opaque-box`. That is, somebody else has put a value in the box, and you don't know what its type is.

If a value constructor can have two different types depending on context, which one are we supposed to write? Historically, we write the universally quantified version, which gives the type in the value context. This convention might have arisen because it can be implemented without changing any of the syntax used to define algebraic data types: if there is a type variable that's not a parameter to the result type, that type variable is considered existentially quantified. That rule is expressed informally as function asX, which is short for "as existential." Here's a simplified specification with just one universally quantified variable α_1 and one existentially quantified variable β_1:

$$asX_1(\forall\alpha_1, \beta_1.\tau_1 \to \alpha_1\ \tau) = \forall\alpha_1.(\exists\beta_1.\tau_1) \to \alpha_1\ \tau.$$

The full version asX handles any number of α_i's and β_i's.

Now that we know about these two different types, what do we do with them? When we have a type like $\forall\alpha.\alpha \to$ `opaque-box`, we know just what to do: substitute any type we like for α. In nondeterministic rules, we nondeterministically substitute exactly the right type; in type inference, we substitute a fresh type variable. Either way, the substitution eliminates the universal quantifier. What about a type like $(\exists\beta.\beta) \to$ `opaque-box`? We would like to do the same thing: eliminate the quantifier and substitute for β. But we can't substitute an arbitrary type, and so we can't substitute a fresh type variable, which, via type inference, might be equated to an arbitrary type. We have to substitute a type that is not only unknown but truly undiscoverable: the hidden type that somebody else put in the box. The name for such a type is a *skolem type,* and the process of substituting skolem types for existentially quantified variables is called *skolemization.*[2]

A skolem type acts a lot like a type constructor: it is equivalent only to itself, and you can't substitute for it during constraint solving. But because a skolem type does not behave in exactly the same way as a type constructor, I use notation that suggests "type constructor" but is not exactly the same: I write a skolem type as $\tilde{\mu}$.

Now I can give typing rules for a value constructor that may appear in two contexts: in an expression or in a pattern. For the expression context, I continue to use the judgment form $\Gamma \vdash K : \tau$, with the same rule as above:

$$\frac{\Gamma(K) = \sigma \qquad \tau' \leqslant \sigma}{\Gamma \vdash K : \tau'}. \tag{VCON}$$

For the pattern context, I define a new judgment form $\Gamma \vdash_p K : \tau$, with a rule that performs these steps:

1. Look up K in Γ to get σ, which is the universally quantified version of K's type.

2. Convert σ to its existentially quantified version.

3. Choose fresh skolem types $\tilde{\mu}_1, \ldots, \tilde{\mu}_m$.

4. Skolemize the existentially quantified type, producing a new type scheme σ'.

[2] Elsewhere you may see the term *skolem variable*; it means the same thing as a skolem type.

5. Instantiate σ' to get τ', the type of K in the pattern context.

The rule looks like this:

$$\frac{\Gamma(K) = \sigma \qquad \mathit{asX}(\sigma) = \forall \alpha_1, \ldots, \alpha_n . (\exists \beta_1, \ldots, \beta_m . \tau_1 \times \cdots \times \tau_m) \rightarrow \tau \\ \{\tilde{\mu}_1, \ldots, \tilde{\mu}_m\} \cap \mathrm{ftc}(\Gamma) = \emptyset \\ \sigma' = (\forall \alpha_1, \ldots, \alpha_m . \tau_1 \times \cdots \times \tau_m \rightarrow \tau)[\beta_1 \mapsto \tilde{\mu}_1, \ldots, \beta_m \mapsto \tilde{\mu}_m] \qquad \tau' \leqslant \sigma'}{\Gamma \vdash_p K : \tau'}.$$

(VconInPattern)

Function ftc (“free type constructors”) finds all the type constructors, including skolem types, used in Γ.

We’re not quite done with skolem types. Skolem types don’t just look different from ordinary type constructors; they are also *semantically* different. An ordinary type constructor like `int` or `bool` always means the same set of values at run time. But a skolem type that appears in a case expression can mean something *different* on *each* evaluation of the case expression. Just think about the `shape` functions above. In `scale-shape`, for example, sometimes the hidden type is `circle`, but other times it is `(list pt)`. But within the scope of the case expression, both of these hidden representations are given the same skolem type, say $\tilde{\mu}_{17}$. But the equivalence $\tilde{\mu}_{17} \equiv \tilde{\mu}_{17}$ is sound only for duration of a single evaluation. So it is absolutely crucial that $\tilde{\mu}_{17}$ not *escape* the case expression. What’s true of $\tilde{\mu}_{17}$ is true of every skolem type:

- A skolem type must not appear in the type of the result.
- A skolem type must not appear in the type of the scrutinee.
- A skolem type must not appear in a constraint in such a way that it wants to be substituted for a type variable that appears free in the environment.

All these means of escape must be prevented.

Lest they escape, the skolem types that are introduced by a pattern match must not appear in either the argument type or the result type of that pattern match.

$$\frac{C, \Gamma, \Gamma' \vdash p : \tau \qquad C', \Gamma + \Gamma' \vdash e : \tau' \\ \theta(C \wedge C') \equiv \mathbf{T} \\ \mathrm{fs}(\theta\Gamma') \cap \mathrm{fs}(\theta\Gamma) = \emptyset \qquad \mathrm{fs}(\theta\Gamma') \cap \mathrm{fs}(\theta(\tau \rightarrow \tau')) = \emptyset}{C \wedge C', \Gamma \vdash \texttt{[}p\ e\texttt{]} : \tau \rightarrow \tau'} \quad \text{(ExistentialChoice)}$$

Function fs finds the (free) skolem types that appear in an environment.

This book ships with two versions of the μML interpreter: interpreter `uml` runs plain μML, and interpreter `umlx` runs μML extended with existential types. The code for the extensions appears in Appendix S.

C.2 GADTs

GADTs, which are short for *generalized algebraic data types*, allow you to attach extra type information to constructed values. The extra type information can help the compiler remove run-time overhead and rule out certain run-time errors. It can also help you build functions that effectively dispatch on the type. GADTs are an advanced language feature, and type inference for GADTs is very involved—too much for me to implement in a bridge language. But in this section I show one example of GADTs, written in the popular functional language Haskell. At the end of the section I mention several other applications.

My main example is a simple evaluator with *tagged values*, which works just like the `eval` functions in this book. In deference to common Haskell style, I write value constructors with only an initial capital letter, not in all capitals as Standard ML programmers do.

S33a. ⟨*transcript* S33a⟩≡ S33b ▷

```
-> (data * value
     [Bool : (bool -> value)]
     [Int  : (int  -> value)])
value :: *
Bool : (bool -> value)
Int : (int -> value)
-> (Bool #t)
(Bool #t) : value
-> (Int 7)
(Int 7) : value
```

The values I can represent include integers and Booleans, and they are distinguished by the value constructors `Int` and `Bool`, which act as *tags*.

Now I can design a little language of expressions, which contains literals, addition, comparison, and conditional:

S33b. ⟨*transcript* S33a⟩+≡ ◁ S33a S33c ▷

```
-> (data * exp
     [Lit  : (value -> exp)]         ;; bool or int
     [Plus : (exp exp -> exp)]       ;; add two ints to make an int
     [Less : (exp exp -> exp)]       ;; compare two ints to make a bool
     [If   : (exp exp exp -> exp)])  ;; look at a bool and choose an 'a
```

This representation is like the representations used throughout this book, and if I use it to write an evaluator, here are some of the things that cost extra or can go wrong:

- Each literal-value expression pays the cost of *two* tags: one from `exp` that marks it as a literal, and one from `value` that marks it as `int` or `bool`.
- Evaluating `Plus` will fail if either argument is a Boolean. Even if the child of a `Plus` node is a `Plus` or a literal `Int`, the evaluator still has to check at run time. Similar checks are implemented in interpreters for μScheme and μML, for example, and if the check fails, an interpreter raises `RuntimeError` or `BugInTypeInference`.
- I know that evaluating `Plus` produces an `int` and evaluating `Less` produces a `bool`, but I have no way to tell the compiler. And nothing stops me from creating terms that I know *can't* be evaluated:

  S33c. ⟨*transcript* S33a⟩+≡ ◁ S33b

  ```
  -> (val ill-typed (Plus (Less (Lit (Int 2)) (Lit (Int 9))) (Lit (Int 1))))
  (Plus (Less (Lit (Int 2)) (Lit (Int 9))) (Lit (Int 1))) : exp
  ```

For this very simple language, I could work around the problem by defining *two* forms of expression, say `int-exp` and `bool-exp`, which evaluate to integers and Booleans respectively. Value constructors `Plus` and `Less` belong only to `int-exp`, but constructors `Lit` and `If` are polymorphic and have to be duplicated. If I want to add more types, and if I want more polymorphic language constructs, such as `let` expressions and function calls, this trick doesn't scale.

What I'd like to do is use the type system of the *implementation* language (μML, Standard ML, or Haskell) to accomplish two goals:

- Prevent anyone from constructing a term like `ill-typed`, which causes a run-time error if evaluated.
- Explain to the compiler that when *deconstructing* a term, errors are not possible.

The first goal can be addressed using *phantom types*. The second requires GADTs.

Ruling out ill-typed expressions using phantom types

A phantom type is a type parameter that is used to enforce some invariant, but that does not actually appear in a representation. Enforcing the invariant requires type constraints on functions, and often these functions are "smart constructors." Unfortunately I can't express type constraints in μML—adding them is Exercise 1 at the end of this appendix. I could do the examples in Standard ML, but for coherence with the rest of the section, I switch to Haskell, which supports not only type constraints but also GADTs.

In Haskell, a type constructor is written with a capital letter, and a type variable is written with a lower-case letter. The same rules apply to value constructors and value variables; the design is very consistent, but it is sometimes difficult to distinguish the type language from the term language. Here again are the definitions of types `value` and `exp` from above, written in in Haskell:[3]

S34a. ⟨*Haskell definitions for GADT example* S34a⟩≡ S34b ▷

```
data Value :: * where
  Int  :: (Int  -> Value)
  Bool :: (Bool -> Value)

data Exp :: * where
  Lit  :: (Value -> Exp)
  Plus :: (Exp -> Exp -> Exp)
  Less :: (Exp -> Exp -> Exp)
  If   :: (Exp -> Exp -> Exp -> Exp)
```

Notice the double colons. They are used in the term language to say that a value has a given type, and they are used in the type language to say that a type has a given kind. Also, Haskell has no multi-argument functions or value constructors, so the value constructors are Curried.

As in μML, I can make nonsensical values of type `Exp`. To rule them out, I take two additional steps: First, I define `TypedExp`, which takes a phantom type parameter. A `TypedExp` wraps an `Exp`; the `newtype` definition guarantees that `Exp` and `TypedExp` have the same representation, and that applying or matching on value constructor `TE` costs nothing at run time.

S34b. ⟨*Haskell definitions for GADT example* S34a⟩+≡ ◁ S34a S35a ▷

```
newtype TypedExp :: * -> * where
  TE :: forall a . Exp -> (TypedExp a)
```

Second, I define *smart constructors* for `TypedExp`. These constructors are constrained by *type signatures*, so any value made using them represents a well-typed expression. A type signature acts like a `check-type`, only stronger: it permits the

[3] If you have experience with Haskell, you should be horrified by all the parentheses. The parentheses are for inexperienced readers; they make the Haskell code look more like μML code.

function to be used *only* at instances of the specified type. (In Exercise 1, you can add a similar form, type-is, to μML.)

S35a. ⟨*Haskell definitions for GADT example* S34a⟩+≡ ◁S34b S35c▷

```
int  :: (Int  -> (TypedExp Int))
bool :: (Bool -> (TypedExp Bool))
plus :: ((TypedExp Int) -> (TypedExp Int) -> (TypedExp Int))
less :: ((TypedExp Int) -> (TypedExp Int) -> (TypedExp Bool))
ifx  :: (forall a .
          ((TypedExp Bool) -> (TypedExp a) -> (TypedExp a) -> (TypedExp a)))

int  n = (TE (Lit (Int  n)))
bool b = (TE (Lit (Bool b)))
plus (TE e1) (TE e2)         = (TE (Plus e1 e2))
less (TE e1) (TE e2)         = (TE (Less e1 e2))
ifx  (TE e1) (TE e2) (TE e3) = (TE (If e1 e2 e3))
```

Now I can revisit the ill-typed example above. With the smart constructors, the type checker won't let me add a Boolean expression to an integer expression.

S35b. ⟨*GHCI transcript* S35b⟩≡ S36c▷

```
*Bookgadt> (plus (less (int 2) (int 9)) (int 1))

<interactive>:3:8:
    Couldn't match type ‘Bool’ with ‘Int’
    Expected type: TypedExp Int
      Actual type: TypedExp Bool
    In the first argument of ‘plus’, namely ‘(less (int 2) (int 9))’
    In the expression: (plus (less (int 2) (int 9)) (int 1))
*Bookgadt>
```

Unfortunately, the eval function still has to account for the possibility of error at run time:

S35c. ⟨*Haskell definitions for GADT example* S34a⟩+≡ ◁S35a S36a▷

```
eval :: TypedExp a -> Value
eval (TE e) =
 let ev e =
       case e of
         { (Lit v)       -> v
         ; (Plus e1 e2)  -> case (ev e1, ev e2) of
                               { (Int n, Int m) -> (Int (m + n))
                               ; _ -> (error "expected integers")
                               }
         ; (Less e1 e2)  -> case (ev e1, ev e2) of
                               { (Int n, Int m) -> (Bool (m < n))
                               ; _ -> (error "expected integers")
                               }
         ; (If e1 e2 e3) -> case (ev e1) of
                               { (Bool b) -> (ev (if b then e2 else e3))
                               ; _ -> (error "expected Boolean")
                               }
         }
 in  ev e
```

Smart constructors buy you a lot, and if you're stuck programming in μML, Standard ML, or standard Haskell, keep them in mind. But if you're lucky enough to be programming in OCaml, extended Haskell, Agda, or Idris, you can use GADTs instead.

Eliminating tags using GADTs

A GADT is a *generalized* algebraic data type. What's generalized? The types of the value constructors. In particular, GADTs lift the restriction that the type parameters passed to the result type must be type variables. In a GADT, you can use any type as a type parameter. In our running example, instead of wrapping `Exp` in `TypedExp`, I just define `TExp`, with these value constructors:

S36a. ⟨*Haskell definitions for GADT example* S34a⟩+≡ ◁S35c S36b▷

```
data TExp :: * -> * where
  TLit  :: forall a . (a -> (TExp a))
  TPlus :: ((TExp Int) -> (TExp Int) -> (TExp Int))
  TLess :: ((TExp Int) -> (TExp Int) -> (TExp Bool))
  TIf   :: forall a . ((TExp Bool) -> (TExp a) -> (TExp a) -> (TExp a))
```

The `TLit` and `TIf` constructors pass type variable `a` to `TExp`, but `TPlus` and `TLess` pass type parameters `Int` and `Bool`, respectively.

The definition of `TExp` displays a number of pleasing properties:

- The `Value` type is gone. The `TLit` constructor is polymorphic, which means we can take a value of *any* type `a` and turn it into an expression.
- We know that `TPlus` expects integer expressions and returns an integer expression. `TLess` expects integer expressions and returns a Boolean expression.
- `TIf` is polymorphic: the condition has to be a Boolean expression, but the true and false branches can be expressions of any type, as long as they're the same.

We can also write a new *evaluator* without `Value`. If we evaluate a typed expression of type `(TExp a)`, what we get back is just an `a`. No tags, and no possibility of run-time error:

S36b. ⟨*Haskell definitions for GADT example* S34a⟩+≡ ◁S36a

```
teval :: (forall a . ((TExp a) -> a))
teval e = case e of
    { (TLit a)       -> a
    ; (TPlus e1 e2)  -> ((teval e1) + (teval e2))
    ; (TLess e1 e2)  -> ((teval e1) < (teval e2))
    ; (TIf e1 e2 e3) -> (teval (if (teval e1) then e2 else e3))
    }
```

In this evaluator, results are untagged. Depending on context, function `teval` returns an integer, a Boolean, or a value of unknown type, and we never need a run-time case expression to figure out which is which. For example, the result of evaluating a `TPlus` expression can be passed directly to `+` without any run-time checks. The code is simpler, cleaner, and just works. Here's some evidence:

S36c. ⟨*GHCI transcript* S35b⟩+≡ ◁S35b

```
*Bookgadt> (teval (TPlus (TPlus (TLit 2) (TLit 9)) (TLit 1)))
12
*Bookgadt> (teval (TIf (TLess (TLit 2) (TLit 9)) (TLit "smaller") (TLit "??")))
"smaller"
```

Getting these great results requires some sophisticated type inference, which is well beyond the scope of this book. In 2015, the Glasgow Haskell Compiler started using the "OutsideIn" algorithm, which works type information from the signature of `teval` (the "outside") to the right-hand sides of the choices in the case expression. If you want to try similar examples yourself, remember that to make OutsideIn work, the top-level type signature on `teval` is required.

GADTs are a powerful tool for encoding dynamic properties in static types. In my own work, for example, we use GADTs to represent control-flow graphs in an optimization library; the GADTs govern exactly what code fragments can be composed in sequence, and they guarantee that a finished control-flow graph never contains a dangling edge.

GADTs are used in many contexts to eliminate tags on inputs or outputs. Two of my favorite examples are using GADTs to implement a type-safe version of `printf`, without tags, and using GADTs to represent the stack in an LR parser, which is much like the `ParserState` in Section G.3 (page S196).

GADTs have also been used to encode permissions, and they have been used in many kinds of type-directed computation, including converting values to bit strings and back.

C.3 Further reading

Algebraic data types were first extended to include existentially quantified value constructors by Perry (1991), and the underlying type theory was perfected by Läufer and Odersky (1994). Läufer and Odersky crafted their language to minimize the number of syntactic forms and the number of rules in the type theory, which makes it look very different from the case expressions and patterns we use today. Also, they explain type inference using explicit substitutions, not constraints. If you want additional context for the use of existential types to hide representations, Mitchell and Plotkin (1988) go deep into the type theory, and they also present many programming examples.

GADTs exploded onto the programming-language scene in the early 2000s. My favorite introduction is the book chapter by Hinze (2003), who presents GADTs as an extension of phantom types. Pottier and Régis-Gianas (2006) present an excellent application: they use GADTs to replace an unsafe parsing stack—used by Yacc, Bison, and other parser generators—with a safe, typed data structure. The unsafe stack is essentially the same as the sequence of components used in the C parsers described in Appendix G. My own application of GADTs to a dataflow-optimization library is described by Ramsey, Dias, and Peyton Jones (2010).

Type inference for GADTS has proven challenging; using a GADT's value constructor brings additional type-equality constraints into play, but those constraints apply only on the right-hand side of a choice in a case expression, not more broadly as we are used to. Some good inference algorithms have been proposed, but truly simple, clear explanations of the best algorithms have yet to be written. To get started, I recommend the OutsideIn paper by Schrijvers et al. (2009), but with caveats: the paper describes several different languages and type systems, and you may have trouble understanding the distinctions and relations among them. You may also be overwhelmed by the sheer detail required. A later, less dense version of this paper appeared in a journal (Vytiniotis et al. 2011), but the later treatment is much more abstract. If you already understand the algorithms, you will like the abstraction, but if not, you may find the abstract treatment hard to learn from.

C.4 Exercise

1. *Type constraints.* If you want to define smart constructors that use phantom types, you need a way to constrain a function to be used at a less general type than its implementation permits. Extend μML with a new definition form

   def ::= `(type-is` *value-variable-name type-exp*`)`

   The form is typically used with a function f; you write `(type-is` f σ`)`, and thereafter, f may be used only at the given type scheme, which may be strictly less general than its given type scheme. You check that the claimed type scheme is an instance of f's current type scheme, then update the type environment:

   $$\frac{\Delta \vdash t \rightsquigarrow \sigma :: * \qquad \Gamma(f) = \sigma' \qquad \sigma \leqslant \sigma'}{\langle \texttt{(type-is}\ f\ t\texttt{)}, \Gamma\rangle \rightarrow \Gamma\{f \mapsto \sigma\}}. \tag{TypeIs}$$

   You will reuse the `txTyScheme` function from chunk S456c, and you will find code for $\sigma \leqslant \sigma'$ as part of the implementation of `check-type`.

   A `type-is` definition must follow the definition of the name it constraints. It's not as convenient as `check-type` or a Haskell type signature, but it's more convenient than anything you can write in Standard ML.

Appendix D contents

Prolog and logic programming D

> *The validity of the processes of analysis does not depend upon the interpretation of the symbols which are employed, but solely upon the laws of their combination. … We might justly assign it as the definitive character of a true Calculus, that it is a method resting upon the employment of Symbols, whose laws of combination are known and general, and whose results admit of a consistent interpretation. . . . It is upon the foundation of this general principle, that I purpose to establish the Calculus of Logic.*
>
> George Boole (1847), *The Mathematical Analysis of Logic*

In the main text, both operational semantics and type systems are written using inference rules. But inference rules are good for more than just *describing* a programming language; inference rules can *be* a programming language. And they form a programming language on a completely different model: instead of evaluating expressions to produce a value, a *logic-programming* language looks for a way to instantiate variables to create a provable judgment. The most venerable such language is Prolog.

Prolog once enjoyed wide use, and several people who have used this book have asked for a chapter on Prolog. But Prolog doesn't quite work as a chapter, for two reasons:

- I can't provide an interpreter that performs well. A modern interpreter is so much faster than my μProlog interpreter that it's not a question of getting answers more slowly; there are many problems for which a modern Prolog provides answers instantly, where μProlog won't terminate in your lifetime.

- If you want to use traditional Prolog as it was used in the 1990s, and especially if you want to understand the operational interpretation and the cut, I can help you a lot. But logic programming has moved on, and if you want to use modern methods of logic programming, I can't help you.

Because there has been significant demand for it, I haven't abandoned my work on μProlog. But because it's neither fast nor up to date, it is relegated to this appendix.

Prolog got created not because its designers were keen to use inference rules, but because inference seemed like a way to address the problem of machine intelligence. In Alan Turing's famous test, a machine is deemed intelligent if it can converse in a way that is indistinguishable from human. And any such machine must show some ability to reason about facts. Such reasoning was central to research that produced the first computer programs you could converse with, which were written in the late 1960s and early 1970s.

Reasoning itself has been a topic of study since ancient Greece. The best-known ancient work is probably Aristotle's *Organon.* You may have seen this example of "syllogism":

> All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

The important thing is the *form* of the argument, not the meanings of the nouns and adjectives. It is equally valid to say,

> All rabbits are mammals. Bugs Bunny is a rabbit. Therefore, Bugs Bunny is a mammal.

The content is not so convincing, but the form is the same. Today we would express only the form, using mathematical abstraction:

> I claim $\forall X : p(X) \implies q(X)$.
> I claim $p(a)$.
> Therefore, I conclude $q(a)$.

All these examples embody the same reasoning. The formal study of this sort of reasoning—*mathematical logic*—is about form (syntax), not content ("models" or "interpretations").

Mathematical logic took on its modern form in the 19th century. Logical reasoning was formulated algebraically by George Boole in 1847, whom we honor with our "Booleans." But the most important single advance in the study of rigorous reasoning was Gottlob Frege's *Begriffsschrift,* or "concept notation," published in 1879. Frege not only put prior notations into a satisfying uniform framework; he also invented quantifiers and bound variables. His notation is strangely two-dimensional, and it involves a bewildering variety of fonts, but it is modern logic.

Mathematical logic is used throughout the theory of programming languages. Mathematical logic developed judgments, syntactic proofs, inference rules, and valid derivations—in other words, all of our operational semantics and type theory. Problems in logic inspired Alonzo Church to invent the lambda calculus, as a way of studying free and bound variables. And lambda calculus led to Lisp, Scheme, and to other functional languages.

Using logic to reason about programming languages is great, but this chapter presents a different development: logic itself can *be* a programming language. The foundations for this idea were laid in the late 1960s and early 1970s, as first-order logic was being applied to many problems whose solutions might lead to machines that could be called intelligent. The foremost such problems lay in *automated theorem proving* and in *man-machine communication.* And by the early 1970s, simple communication in natural language was no longer the sole province of science-fiction writers. As an example, here is my translation of a dialog with an early system developed by Alain Colmerauer (1973) and his colleagues at the university of Aix-Marseille. The user's entries are in Roman type and the system's responses are in italics:

> Every psychiatrist is a person.
> Each person he analyzes is sick.
> Jacques is a psychiatrist in Marseille.
> Is Jacques a person?
>   *Yes.*
> Where is Jacques?
>   *In Marseille.*
> Is Jacques sick?
>   *I don't know.*

A key part of this system was a new programming language designed to simplify the programming of logical inference based on predicates. This language, Prolog, was invented by Colmerauer and his team. Prolog, which stands for "*pro*gramming in *log*ic," remains the best-known and most popular logic-programming language.

In Prolog, you solve a problem not by giving a computational procedure, but by stating a predicate that must be true of any correct answer, along with logical axioms and inference rules that can be used to prove such a predicate. If you understand how the proof engine works, you can craft your logic in such a way that when you ask about a predicate, out pop values that make it provable—and those values solve your problem. The programming techniques you need and the workings of the proof engine are described below.

D.1 Thinking in the language of logic

In functional programming, we *define* functions: a function's behavior is specified by a body we write. In logic programming, we don't define functions; functions are *unspecified*. Instead we define *predicates* that give properties of the results of applying functions, or properties of mathematical objects, or relationships among any of these.

In functional programming, we get values by applying functions to other values. In logic programming, we get values by asking if there are any values that make a given proposition provable. This computational model is so different from the model found in most programming languages that unless you are already trained in mathematical logic, you are likely to find it strange. The notation *looks* like it is applying functions to variables or to the results of applying other functions, but the names that look like functions and variables don't behave the way we expect functions and variables to behave. To write logic programs that work, you need to keep in mind what kinds of things the names in a program are actually standing for. To begin, let's look at names in the language of logic.

Atoms and objects

Prolog refers to mathematical objects by name; an object is named by an *atom*. Examples of atoms include `jacques`, `marseille`, `elizabeth`, `charles`, `z`, `table`, `smallmouth`, and `stephen_hawking`. These atoms are also Scheme atoms. Prolog uses the same word as Scheme for the same reason: an atom can't be taken apart. All Prolog knows about an atom is that an atom is identical to itself—plus whatever facts about the atom we choose to share. What Prolog knows about an atom is exactly what mathematical logic knows about an unspecified object.

Prolog also treats numbers as objects. It can even do a little arithmetic.

Functors

Where mathematical logic works with "unspecified function symbols," Prolog works with *functors*.[1] The opening dialog about Jacques the psychiatrist is so simple that there are no functors, but in the theory of lists, `cons` is a functor, and in Peano's theory of the natural numbers, `s` (successor) is a functor. As further examples, Section D.6 below talks about moving blocks on a table, and it uses functors `on` and `move`. And Section D.7 uses Prolog data to represent Scheme programs, and in that setting, `lambda` and `apply` are functors.

[1] "Functor" is regrettable word. It is important in Prolog, in Standard ML, in Haskell, and in category theory—and in each context, it means something different. At least there is an analogy between the Haskell meaning and the category-theoretic meaning.

In mathematical logic, functors and atoms are the same kind of thing: unspecified functions. An atom is just an unspecified function of zero arguments: a constant.

Terms

If the idea is to prove facts about properties and relations, what sorts of things have properties? What sorts of things can be related? *Terms.* All Prolog data (and in full Prolog, also Prolog code) can be represented as terms. "Term" is a recursive data type that is analogous to S-expression in Scheme, and like S-expressions, terms can be defined inductively. A term is one of the following:

- An atom
- A number
- A functor applied to one or more terms
- A *logical variable* (discussed below)

Here are some examples of terms:

- Term `cons(0, cons(1, cons(2, nil)))` represents a list containing the first three natural numbers.
- Term `s(s(s(z)))` represents the natural number 3, as it is axiomatized in Peano's system.
- Term `move(b, table)` represents the action of moving block `b` onto a table.
- Term `lambda(cons(x, nil), x)` represents Scheme code for the identity function.

In simple examples, most terms are atoms or lists.

If these ideas seem new or confusing, you can't go wrong with an analogy: the world of Prolog data is like one big algebraic data type.

- An atom is like a nullary value constructor, just like `nil` or `NONE` in ML.
- A functor is like a value constructor that takes one or more arguments, like `SOME` or `cons` in μML.
- A Prolog term is like a value of algebraic data type.

Prolog terms even participate in a form of pattern matching, just like ML values of algebraic data type. Only the concrete syntax is different. (And if you ever use the functional language Erlang, which is an excellent choice for parallel and distributed computing, you'll encounter exactly the same form of data, using Prolog syntax.)

Properties and propositions

A *property* is a thing that can be true of one object, or of one term. In logic, it's a "one-place relation." The properties in the opening dialog are `psychiatrist`, `person`, and `sick`. Example mathematical properties include `natural_number` and `nonzero`. An example property of a list is `null`, and an example property of an ML type (from Section D.7) is `admits_equality`. A property is a thing we can apply to an object or term to get a fact, or to get a proposition that might be a fact.

Example propositions include `psychiatrist(jacques)`, `person(jacques)`, and `sick(stephen_hawking)`. Mathematical examples include `natural_number(z)` and `nonzero(s(s(s(z))))`. Prolog has no type system, so you can also write bizarre propositions like `natural_number(jacques)`, `null(table)`, and `sick(3)`. I hope you define your logical systems so that these propositions are not provable. In μProlog, but not in full Prolog, this sort of thing can be checked:

S45a. ⟨*transcript* S45a⟩≡ S45b ▷

```
?- check_unsatisfiable(natural_number(jacques)).
?- check_unsatisfiable(null(table)).
?- check_unsatisfiable(sick(3)).
```

Relations and predicates (and more propositions)

A *relation* is a thing that can be true of two or more objects. In logic, it's just a "relation." The relations in the opening dialog are `analyzes`, and `is_in`. The only fact given about these relations is `is_in(jacques, marseille)`. Other relations lead to such propositions as `mother(elizabeth, charles)`, `eats(smallmouth, fly)`, and `relatively_prime(12, 35)`.

The distinction between "property" and "relation" may help us think about problems, but Prolog sees properties and relations as the same kind of thing: both are *predicates*. A property is a one-place predicate—that is, a predicate that takes one argument—and a relation is a predicate that takes two or more arguments. We can even imagine zero-place predicates, like the predicates `imokay` and `youreokay` (I'm OK, you're OK) in Section D.3.4. This kind of generalization, where things that appear different are revealed as instances of one kind of more general thing, also happens in mathematical logic.

Syntactically, propositions and terms look exactly the same. So do functors and predicates. The distinctions are a matter of symbolism and intent. A functor symbolizes a way of making a thing from other things; a predicate symbolizes a property of a thing or a relation among things. A term represents a thing you intend to use as data; a proposition represents a statement you intend either to try to prove or to assert as a fact.

These distinctions will help you think, but they are artificial. In practice, propositions are also perfectly good Prolog data. Full implementations of Prolog use the "programs as data" paradigm (Section E.1 on page S117 of Appendix E) just as often and just as effectively as full implementations of Scheme. Two examples of this use, the special primitive predicates `assert` and `retract`, are described in Section D.8.3 below.

Facts, rules, variables, and clauses

Given a proposition, a Prolog programmer can do three things: assert it as a *fact*, assert that it follows from other propositions (a *rule*), or ask if there is a way to prove it (a *query*). Here are some facts that are asserted in the opening dialog:

S45b. ⟨*transcript* S45a⟩+≡ ◁ S45a S46a ▷

```
?- [fact].  /* makes the interpreter ready to receive facts */
-> psychiatrist(jacques).
-> is_in(jacques, marseille).
```

The opening dialog also asserts some rules, such as "every psychiatrist is a person." To express this rule in the language of logic, we need a *logical variable*. I use P.

To write the rule in logic, we say "for every P, if P is a psychiatrist, then P is a person." To write it *formally,* we say

$$\forall P : \mathit{psychiatrist}(P) \implies \mathit{person}(P).$$

This mathematical expression is a "formula" of *first-order logic.* The idea of "formula" is not so important here, but "first-order" is crucial, because it describes a limitation built into Prolog. In first-order logic, *a logical variable may stand for any object or term, but it may not stand for a functor or a predicate.* When you work with Prolog, remember what kind of thing a variable can stand for—just as when you work with Impcore, you remember that a variable can hold a value but not a function.

When we assert a rule to Prolog, we don't simply present a formula in first-order logic. Prolog is limited a particular form of formula called the "Horn clause." Fortunately, you don't need to know what a Horn clause is, because the syntax of Prolog is set up so that you don't write a Horn clause as a formula, you write it as an *inference rule.* A Prolog inference rule is guaranteed to be logically equivalent to a Horn clause, and vice versa (Exercise 11, page S101). In language of inference rules, the rule "every psychiatrist is a person" is written

$$\frac{\mathit{psychiatrist}(P)}{\mathit{person}(P)}.$$

(The universal quantifier $\forall$ has disappeared; it is implicit.) In Prolog, this rule is written as follows:

S46a. ⟨*transcript* S45a⟩+≡ ◁ S45b S46b ▷

```
  -> person(P) :- psychiatrist(P).
```

The conclusion is written on the left, and the premises (here, just one premise) on the right.

As another example, let's formalize the rule "each person [a psychiatrist] analyzes is sick." We should think like logicians:

- What objects are in the problem? A person who is a psychiatrist, and another person who analyzed by the psychiatrist. We don't know the identity of either object, so we use a logical variable to stand for each one. How about `Doctor` and `Patient`? (In Prolog, the name of an atom, functor, or predicate begins with a lowercase letter, and the name of a logical variable begins with an uppercase letter.)

- What properties and relations—that is, what *predicates*—are in the problem? The property `sick` and the relation `analyzes`, both of which are mentioned above.

At this point I hope you could write the rule yourself:

S46b. ⟨*transcript* S45a⟩+≡ ◁ S46a S47a ▷

```
  -> sick(Patient) :- psychiatrist(Doctor), analyzes(Doctor, Patient).
```

The facts about `psychiatrist` and `is_in` and the rules about `person` and `sick` capture the knowledge of the first three lines of the opening dialog. Before we go on to the queries, let's observe that facts and rules are similar: both are assertions about the world. And just as Prolog considers properties and relations to be special cases of one kind of thing—predicates—so does it also consider facts and rules to be special cases of one kind of thing: *clauses.* (A fact is sometimes also called an *axiom,* especially if the fact includes logical variables, but such a fact is just another form of clause.) A Prolog "program" is just a sequence of clauses, each one of which is either a fact or a rule. In an implementation of Prolog, the sequence can be represented in a more sophisticated way, called a *database.*

Queries

Once we have a database, we can ask questions about it. A question, called a *query*, is a proposition that might or might not be provable using the facts and rules we have at hand. Prolog will try to find out. Is Jacques a person?

S47a. ⟨*transcript* S45a⟩+≡ ◁S46b S47b▷

```
-> [query].  /* makes the interpreter ready to answer queries */
?- person(jacques).
yes
```

A more interesting query is one that includes logical variables. In Prolog, we cannot ask "where is Jacques?" What we ask instead is "is there a location L such that Jacques is in L?"

S47b. ⟨*transcript* S45a⟩+≡ ◁S47a S47c▷

```
?- is_in(jacques, L).   /* where is Jacques? (as close as Prolog comes) */
L = marseille
yes
```

When we present a query like `is_in(jacques, L)`, what we are really asking if there is any term we can substitute for the logical variables such that the resulting proposition is *provable*. (Just like mathematical logic, logic programming deals in provability, not truth.) A Prolog system is not as sophisticated as the language-processing system shown in the open dialog. When asked if Jacques is sick, Prolog can't prove it, so it answers "no."

S47c. ⟨*transcript* S45a⟩+≡ ◁S47b S47d▷

```
?- sick(jacques).
no
```

D.2 Using Prolog

A Prolog program can involve atoms, objects, functors, terms, properties, relations, predicates, facts, rules, and clauses. To illustrate these words and the ideas behind them, this section uses Prolog to solve two small problems.

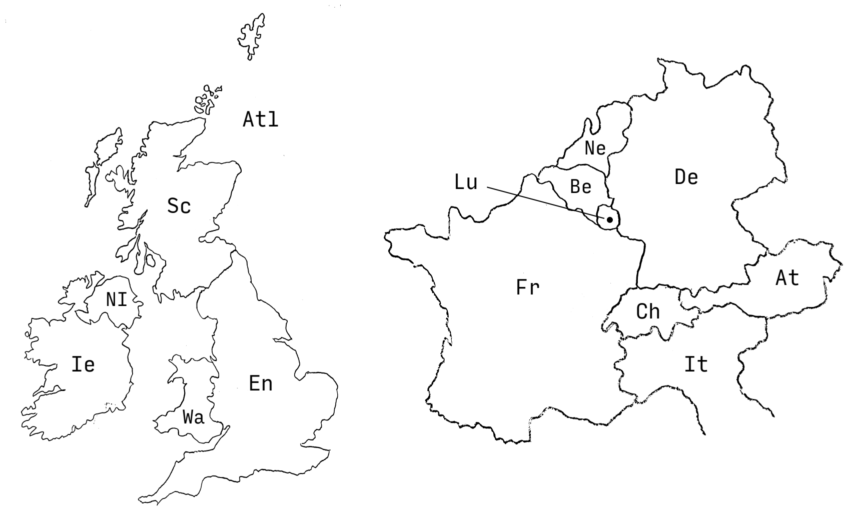
Small example: Map coloring

It is an old problem to ask how many colors are needed to color a map of political jurisdictions in such a way that when two jurisdictions are adjacent, they get different colors. The fact that four colors always suffice is one of the first interesting theorems to be proved with the aid of a computer. In this section, I color a map with *three* colors. A coloring is expressed by substituting colors for logical variables.

In my model, the mathematical objects are colors; I use `yellow`, `blue`, and `red`. To express the key constraint, the colors of adjacent jurisdictions must be different, I introduce the notion of "difference," which is a relation between two colors. The predicate `different` may be proved by any of the following facts:

S47d. ⟨*transcript* S45a⟩+≡ ◁S47c S48▷

```
-> [fact].  /* makes the interpreter ready to receive facts */
-> different(yellow, blue).
-> different(blue, yellow).
-> different(yellow, red).
-> different(red, yellow).
-> different(blue, red).
-> different(red, blue).
```



(a) The British Isles

(b) Part of Western Europe

Figure D.1: Maps

I have to say not only blue is different from red but also that red is different from yellow; Prolog can't tell that I intend `different` to be a symmetric relation.

Now let's use the `different` predicate to color the map of the British Isles shown in Figure D.1a on the current page. To convert the map-coloring problem into a problem in formal logic, I state what relations must hold among the colors of a properly colored map. I obtain the relations by looking at each country and seeing what countries both adjoin it and follow it in the list. For purposes of this problem, the Atlantic Ocean is a country, so map (a) is properly colored by colors `Atl`, `En`, `Ie`, `NI`, `Sc`, and `Wa` if and only if the following predicates hold:

- Color `Atl` is different from `En`, `Ie`, `NI`, `Sc`, and `Wa`.
- Color `En` is different from `Sc` and `Wa`
- Color `Ie` is different from `NI`

There are an awful lot of predicates, so I want to abstract them away into a single predicate `britmap_coloring(Atl, En, Ie, NI, Sc, Wa)`, which means that colors `Atl` through `Wa` constitute a proper coloring of map (a). I do so by giving Prolog an inference rule:

S48. ⟨*transcript* S45a⟩+≡ ◁S47d S49a▷

```
-> britmap_coloring(Atl, En, Ie, NI, Sc, Wa) :-
     different(Atl, En), different(Atl, Ie), different(Atl, NI),
     different(Atl, Sc), different(Atl, Wa),
     different(En, Sc), different(En, Wa),
     different(Ie, NI).
```

This rule should be read as saying

> The colors *Atl* to *Wa* constitute a proper coloring of map D.1a if *Atl* is different from *En*, *Atl* is different from *Ie*, *Atl* is different from *NI*, and so on.

If it were a rule of type theory or operational semantics, we would write it this way:

$$\frac{\begin{array}{ccc} & \texttt{different}(Atl, En) \quad \texttt{different}(Atl, Ie) & \\ \texttt{different}(Atl, NI) & \texttt{different}(Atl, Sc) & \texttt{different}(Atl, Wa) \\ \texttt{different}(En, Sc) & \texttt{different}(En, Wa) & \texttt{different}(Ie, NI) \end{array}}{\texttt{britmap\_coloring}(Atl, En, Ie, NI, Sc, Wa)}$$

Here is the corresponding rule for a fragment of map (b), which is itself a fragment of a map of Europe:

S49a. ⟨*transcript* S45a⟩+≡ ◁S48 S49b▷

```
-> fragment_coloring(Be, De, Fr, Lu) :-
     different(Be, De), different(Be, Fr), different(Be, Lu),
     different(De, Fr), different(De, Lu),
     different(Fr, Lu).
```

The clauses in the database model the two map-coloring problems. To find out what propositions Prolog can prove from these clauses, we issue *queries*. For example, we can ask if simply rotating colors results in a valid coloring of map (a):

S49b. ⟨*transcript* S45a⟩+≡ ◁S49a S49c▷

```
-> [query].  /* makes the interpreter ready to answer queries */
?- britmap_coloring(yellow, blue, red, yellow, blue, red).
no
```

The query is a proposition, and the interpreter responds that no, it can't prove this proposition.

So far, so good. But not very useful. What we would really like to know is *do there exist* colors A to F such that map (a) is properly colored? In Scheme, we would have to write a function that takes a map as argument and returns the colors as results. But logic programming is not about functions; it's about relations. And we can ask if colors exist by posing a query that asks about a relation among *logical variables*.

In Prolog, any identifier beginning with a capital letter is a logical variable, and when given a query that relates logical variables, the Prolog engine searches for values of the logical variables such that the query can be proved. Such a query is called a *goal*. For example, we ask for a coloring of map (a) by issuing this query:

S49c. ⟨*transcript* S45a⟩+≡ ◁S49b S49d▷

```
?- britmap_coloring(Atl, En, Ie, NI, Sc, Wa).
Atl = yellow
En = blue
Ie = blue
NI = red
Sc = red
Wa = red
yes
```

Prolog found a coloring. It not only reports back that the query can be satisfied; it also provides a *satisfying assignment* to the logical variables. When there is no satisfying assignment, Prolog reports as follows:

S49d. ⟨*transcript* S45a⟩+≡ ◁S49c S50▷

```
?- fragment_coloring(Be, De, Fr, Lu).
no
```

Interacting with the interpreter

The example above shows an unusual property of our μProlog interpreter: it has two *modes*. In *rule mode*, the prompt is `->`, and the interpreter silently accepts facts or rules. In *query mode*, the prompt is `?-`, and the interpreter answers queries based on the facts known to it. Entering "`[query].`" puts the μProlog interpreter into query mode. Entering "`[rule].`" or "`[fact].`"[2] puts it into rule mode.

This odd style of interaction is necessary because Prolog uses the *same* concrete syntax for both queries and facts. Other implementations of Prolog also use modes. We could get rid of the modes by using nonstandard syntax, but then you wouldn't be able to use the example code with other Prolog interpreters. And for some problems, you need another Prolog interpreter—μProlog can be too slow.

Naming predicates

Unlike a function name in ML, Impcore, or μScheme, a predicate symbol in Prolog can be used with any number of arguments. A predicate is identified with a *combination* of its symbol and an *arity,* which is the number of arguments used with the symbol. The predicates used in the map-coloring example are `different/2`, `britmap_coloring/6`, and `fragment_coloring/4`. The same symbol may be used at more than one arity; two predicates with the same symbol but different arities are different predicates.

To illustrate the importance of arity in defining predicates, Wolf (2005) points out that in English, "married" can be either a one-place predicate or a two-place predicate. The two-place predicate says that two people are married to each other. The one-place predicate says that a person is married to some other person, the identity of whom is not stated. Each person an a marriage is individually married, and we can say so in Prolog:

S50. ⟨*transcript* S45a⟩+≡ ◁ S49d S51a ▷

```
?- [fact].
-> married(X) :- married(X, Y).
-> married(Y) :- married(X, Y).
```

Wolf (2005) tells a story about an adulterous couple who check into a motel. The clerk is a bluenose who asks, "are you two married?" The clerk means to ask

```
married(adulterer1, adulterer2)
```

which, in Wolf's story at least, isn't true. But the informal English can also mean

```
married(adulterer1), married(adulterer2)
```

which, in Wolf's story, is true. The couple check in successfully, but the sequel involves an indictment for perjury. That's the difference between `married/1` and `married/2`.

Second small example: Lists and list membership

As a second example of programming in Prolog, let's see how Prolog computes with lists. Just as in μScheme and μML, a list is either empty or is made by applying `cons` to an element and a list. In μProlog, the empty list is represented by the atom `nil`. Symbols `cons` and `nil` are respectively a functor and an atom, but think of them as unspecified function symbols (`nil` is a function of zero arguments). They act like value constructors.

Other implementations of Prolog may use symbols other than `nil` and `cons`, but fortunately, Prolog's lists are normally written using syntactic sugar. The empty list is "`[]`," a cons cell is `[`x`|`xs`]`, and the list of elements a to z is `[`a`,`b`,...,`z`]`. There is also a more rarely used form; `[`a`,`b`,...,`y`|`zs`]` gives initial elements and a tail zs

[2] Or "`[user].`" or "`[clause].`" Don't ask.

of unknown length; it stands for cons(a, cons(b, cons(..., cons(y, zs)))). This sweet, sugary syntax is compatible with any implementation.

Now that we know how to write a list, how do we test for membership? In μScheme or μML, we would write a function. But in Prolog, membership is a predicate, not a function. Predicate member(x, xs) should be provable if and only if value x is a member of list xs. What do we know about membership? That x is not a member of the empty list, and x is a member of a nonempty list if it is the head or if it is a member of the tail. In the language of evidence and proof,

- If xs has the form [x | ys], for any list ys, then that's sufficient evidence to prove member(x, xs).
- If xs has the form [y | ys], for any y and ys, and if member(x, ys) is provable, then that's sufficient evidence to prove member(x, xs).
- No other evidence would justify a claim of member(x, xs).

This reasoning can be captured in a tiny proof system:

$$\frac{}{\texttt{member}(x, \texttt{[}x \,|\, ys\texttt{]})} \qquad\qquad \frac{\texttt{member}(x, ys)}{\texttt{member}(x, \texttt{[}y \,|\, ys\texttt{]})}$$

Each rule of this system can be expressed as a Prolog clause:

S51a. ⟨*transcript* S45a⟩+≡ ◁S50 S51b▷

```
?- [rule].
-> member(X, [X|XS]).
-> member(X, [Y|YS]) :- member(X, YS).
```

These clauses, like all Prolog clauses, can be used only to prove goals. That is, they show only where the member predicate holds. When no clause applies, Prolog always considers the goal to be unprovable. Like other forms of logic, Prolog doesn't deal in truth or falsehood; it deals only in *provability*. And Prolog rules are just like rules of operational semantics; they say only when a judgment is provable.

As above, we can use these clauses by making a query involving member:

S51b. ⟨*transcript* S45a⟩+≡ ◁S51a S51c▷

```
-> [query].
?- member(3, [2, 3]).
yes
?- member(3, [2, 4]).
no
```

We can even use a logical variable to ask for a member of a list, or a member satisfying a given predicate:

S51c. ⟨*transcript* S45a⟩+≡ ◁S51b S54a▷

```
?- member(X, [1, 2, 3, 4]).
X = 1
yes
?- member(X, [1, 2, 3, 4]), X > 2.
X = 3
yes
?- member(X, [1, 2, 3, 4]), X > 20.
no
```

This is the idea behind Prolog: you describe a logical predicate that captures the properties of the values you want, and the interpreter searches for values having those properties.

```
clause-or-query  ::= clause | query | mode-change | use | unit-test
clause           ::= goal [:- goals] .
query            ::= goals .
goals            ::= goal {, goal}
goal             ::= term is term
                   | term binary-predicate term
                   | predicate [(term {, term})]
term             ::= atom
                   | functor (term {, term})
                   | term binary-functor term
                   | (term :- term{, term})
                   | [term{, term} [|term]]
                   | [ ]
                   | integer
                   | variable
mode-change      ::= [query]. | [rule]. | [fact]. | [clause]. | [user].
use              ::= [filename].
unit-test        ::= check_satisfiable(goals)
                   | check_unsatisfiable(goals)
                   | check_satisfied(goals {, variable = term})
predicate        ::= ! | name beginning with lower-case letter
binary-predicate ::= name formed from symbols |%^&*-+:=~<>/?`$\
atom, functor    ::= name beginning with lower-case letter
binary-functor   ::= name formed from symbols |%^&*-+:=~<>/?`$\
variable         ::= name beginning with upper-case letter
```

Figure D.2: Concrete syntax of μProlog

D.3 The language

D.3.1 Concrete syntax

The examples above show most of Prolog. Data structures are like the algebraic data types of μML, except there are no types and no type definitions; imagine one big algebraic data type, called *term*. Names like `yellow`, `red`, `cons`, and `nil` act like ML value constructors, and they make terms. But they aren't called value constructors; they're called atoms and functors. Prolog also includes integer data, and full Prolog includes many primitive predicates. The full concrete syntax of μProlog is shown in Figure D.2.

As the figure shows, μProlog is organized differently from the other bridge languages. There are no definitions—μProlog's database is extended by adding *clauses*. A clause doesn't define anything, and μProlog's basis does not include a global environment—the only state maintained at top level is the database of clauses.

When it has no right-hand side, a clause can be called a *fact* or an *axiom*. When a clause does have a right-hand side, it can be called a *rule*. The parts of a rule also have their own names: the left-hand side is the *head* of the rule, sometimes also called the *conclusion* or even the *left-hand side*. The list of phrases following `:-` is the *body*; the individual elements may be called the *premises* or the *subgoals*.

Clauses and queries are formed from *goals*, which are themselves formed from terms. Terms would be analogous to expressions in other languages, provided those expressions were formed using only value constructors, literals, and application. Here are some examples:

```
[14, 7]              mktree(1, nil, nil)
ratnum(17, 5)        on(a, table)
```

These structures are called "terms" rather than "expressions" because Prolog doesn't "evaluate" them. In Prolog, terms do duty as *both* abstract syntax and values. *Functors* like `cons`, `mktree`, and `ratnum` aren't functions, and they don't code for computation; they construct data. Terms can also contain logical variables, which are identifiable as such because a variable starts with a capital letter, as in `[X|XS]` or `on(Block, table)`. If a term or a clause contains no logical variables, it is called *ground*.

μProlog includes some primitive predicates: <, >, >=, and =< for comparing numbers,[3] `atom` for identifying atoms, `print` for printing terms, and `is` for computing with numbers. The primitive predicates are explained in Section D.3.5 (page S68).

It's not just the abstract syntax of μProlog that's different; the concrete syntax is different, too. Why doesn't μProlog use the same parenthesized-prefix syntax as the other bridge languages?

- Lots of interesting Prolog programs require extensive search, and our simple interpreter can't compete with Prolog systems built by specialists. Good systems are freely available, and if we want to write interesting μProlog programs, the programs should run on such systems.

- Prolog really is different: there are no functions, no assignment, no mutable variables, no control, no types, no methods, and no evaluation. Prolog has almost no parallels with other languages, so there is almost no reason to use the same syntax.

  There is one exception: Prolog data is almost exactly the same as μML's algebraic data. It would be pleasant to construct it using the same function-application syntax as in μML. But the ability to run μProlog programs on real Prolog systems is more valuable.

The cost of using a different syntax is not too great. The syntax of μProlog is based on the "Edinburgh syntax," which is also the basis for ISO Standard Prolog. The Edinburgh syntax is simple, easy to learn, and easy to parse. At the abstract level, the Edinburgh syntax is a subset of S-expressions. So it's not as big a departure as it may look.

D.3.2 Unit tests

Like the unit tests in other untyped languages, μProlog's unit tests can check that something works and can also check that something doesn't work. But the details are a little different.

- Test `check_satisfiable(`$g_1, \ldots, g_n$`)` passes if there is a substitution that simultaneously satisfies query $g_1, \ldots, g_n$.

- Test `check_unsatisfiable(`$g_1, \ldots, g_n$`)` passes if there is *no* substitution that simultaneously satisfies query $g_1, \ldots, g_n$.

[3] Prolog is intended primarily for symbolic computation, not for numeric computation, so the left-arrow symbol <= is considered too valuable to use for "less than or equal," which is written =<.

- Test `check_satisfied(`$g_1, \ldots, g_n$`,` X_1 = $t_1, \ldots, X_m$ = t_m`)` gives a query and a substitution that is supposed to satisfy it. The test passes if the query is satisfied by the substitution given, which is $\theta = \{X_1 \mapsto t_1, \ldots, X_m \mapsto t_n\}$. That is, query $\theta(g_1), \ldots, \theta(g_n)$ must be satisfiable. Furthermore, unless one of the t_i's contains a logical variable, each $\theta(g_i)$ must be a ground term, and no additional substitutions should be required to satisfy the query.

A unit test may be entered in either query mode or rule mode—but if you want to use another implementation of Prolog, enter your unit tests in rule mode, where they will be taken for clauses.

Here are some example unit tests about list membership:

S54a. ⟨*transcript* S45a⟩+≡ ◁S51c S54b▷

```
?- check_satisfied(member(X, [2, 3]), X = 2).
?- check_satisfied(member(X, [2, 3]), X = 3).
?- check_unsatisfiable(member(X, [2, 3]), X < 2).
?- check_unsatisfiable(member(X, [2, 3]), X > 3).
```

And here are some more about sick persons and numbers.

S54b. ⟨*transcript* S45a⟩+≡ ◁S54a S56a▷

```
?- check_satisfied(person(jacques)).
?- check_unsatisfiable(sick(jacques)).
?- check_unsatisfiable(sick(3)).
```

D.3.3 Abstract syntax (and no values)

Of all the languages in this book, Prolog has the simplest structure. Unusually, Prolog does not distinguish "values" from "abstract syntax"; both are represented as terms. A term is a logical variable, a literal number, or an application of a functor to a list of terms. (An atom is represented as the application of a functor to an empty list of terms.)

S54c. ⟨*definitions of* `term`*,* `goal`*, and* `clause` *for* μ*Prolog* S54c⟩≡ (S55c) S54d▷

```
datatype term = VAR     of name
              | LITERAL of int
              | APPLY   of name * term list
```

A term can be a functor applied to a list of terms; a goal is a predicate applied to a list of terms. Goals and applications have identical structure.

S54d. ⟨*definitions of* `term`*,* `goal`*, and* `clause` *for* μ*Prolog* S54c⟩+≡ (S55c) ◁S54c S54e▷

```
type goal = name * term list
```

A clause is a conclusion and a list of premises, all of which are goals. If the list of premises is empty, the clause is a "fact"; otherwise it is a "rule," but these distinctions are useful only for thinking about and organizing programs—the underlying meanings are the same. Writing our implementation in ML enables us to use the identifier **:-** as a value constructor for clauses.

S54e. ⟨*definitions of* `term`*,* `goal`*, and* `clause` *for* μ*Prolog* S54c⟩+≡ (S55c) ◁S54d

```
datatype clause = :- of goal * goal list
infix 3 :-
```

At the read-eval-print loop, where a normal language can present a true definition, a μProlog program can either ask a query or add a clause to the database. (The switch between query mode and rule mode is hidden from the code in this chapter; the details are buried in Section V.7.3.) I group these actions into a syntactic category called cq, which is short for *clause-or-query*. It is the Prolog analog of a true definition def.

S55a. ⟨*definitions of* def *and* unit_test *for μProlog* S55a⟩≡ (S55c) S55b ▷

```
datatype cq
  = ADD_CLAUSE of clause
  | QUERY      of goal list
type def = cq
```

μProlog includes three unit-test forms.

S55b. ⟨*definitions of* def *and* unit_test *for μProlog* S55a⟩+≡ (S55c) ◁ S55a

```
datatype unit_test
  = CHECK_SATISFIABLE   of goal list
  | CHECK_UNSATISFIABLE of goal list
  | CHECK_SATISFIED     of goal list * (name * term) list
```

Finally, μProlog shares extended definitions with the other bridge languages.

S55c. ⟨*abstract syntax for μProlog* S55c⟩≡ (S593a)

⟨*definitions of* term*,* goal*, and* clause *for μProlog* S54c⟩
⟨*definitions of* def *and* unit_test *for μProlog* S55a⟩
⟨*definition of* xdef *(shared)* S214b⟩
⟨*definitions of* termString*,* goalString*, and* clauseString S597d⟩

D.3.4 Semantics

For semantic purposes, a Prolog "program" is a list of clauses $C_1, \ldots, C_n$ followed by a query gs, where gs is a list of goals. Both clauses and query may include logical variables. The program is "run" by posing the query, and we hope for one of two outcomes:

- Prolog finds an assignment to the query's logical variables such that the resulting instance of the query is provable.
- Prolog finds that no assignment to the query's logical variables makes the query provable.

These outcomes are accounted for by the *logical interpretation* of Prolog. But the logical interpretation doesn't explain everything: it doesn't say *what* assignment is found, and it doesn't account for the possibility that the query might not terminate. To explain Prolog completely, we need a *procedural interpretation*. The logical interpretation, however, is simpler, more intuitive, and a more helpful guide to designing programs. That's where we begin.

The logical interpretation, informally

In the logical interpretation of Prolog, each clause in the database represents a rule of inference, and Prolog uses the rules to prove goals. (An alternative logical interpretation, which views clauses as logical formulas, not as rules of inference, is presented in Section D.8.2, page S90.) Each clause has the form G :- $H_1, \ldots, H_m$, and it is interpreted as a claim about proof: if we can prove $H_1, \ldots, H_m$, we can prove G. When the clause contains logical variables, then if an assignment values to those variables makes every H_i provable, that assignment also makes G provable. In other words, the clause can be read as a rule of inference:

type name 303

$$\frac{H_1 \quad \cdots \quad H_m}{G}.$$

In the special case $m = 0$, the clause "G." means that for every possible assignment of values to G's variables, the resulting instance of G is provable.

In the logical interpretation, a goal has a predicate that might be satisfied, or in the language of semantics, a judgment that might be provable. To satisfy a goal g, we find values of g's logical variables such that the resulting *instance* of g can be proven using the inference rules given as clauses. In other words, we find a derivation.

For example, the goal member(3, [4, 3]) can be proven using the derivation

$$\frac{\dfrac{}{\texttt{member(3, [3])}}}{\texttt{member(3, [4, 3])}}.$$

The upper inference is an instance of the axiom member(X, [X|XS]), and the lower inference is an instance of the rule member(X, [Y|YS]) :- member(X, YS). These two clauses *define* what we mean by the member predicate, or if you prefer, the member judgment.

In logic, rules are independent, and order doesn't matter. Rules can appear in any order, and in each rule, premises can appear in any order. Each rule is sound on its own, and each is independent of the other and of any other rules. Likewise, in the logical interpretation of Prolog, it doesn't matter where clauses occur or in what order, and within a clause, it doesn't matter in what order the subgoals appear. Logically, these two Prolog clauses describe the same rule of inference:

```
sick(Patient) :- psychiatrist(Doctor), analyzes(Doctor, Patient).
sick(Patient) :- analyzes(Doctor, Patient), psychiatrist(Doctor).
```

In logic, there's no preferred direction of computation. It's not like operational semantics; if you write an evaluation judgment $\langle e, \rho \rangle \Downarrow v$, logic doesn't know you mean e and ρ to be inputs and v to be an output. Logic cares only about provability and substitutions.

To illustrate the lack of a preferred direction, let's return to list membership. If you're programming in Scheme and you write a function call (member? x xs), x and xs are inputs, and the result is a Boolean. But in Prolog, you write a query, and you can provide xs as an input and ask for x as an output: "give me a member of this list."

S56a. ⟨*transcript* S45a⟩+≡ ◁S54b S56b▷

```
-> [query].
?- member(X, [4, 3]).
X = 4
yes
```

Logically, we are asking is "does there exist an X such that member(X, [4, 3])?" The answer is "yes," and Prolog exhibits such an X.

According to the logical interpretation of Prolog, you can choose *any* parts of a predicate as inputs and any parts as outputs. For each input, you write a term, and for each output, you write a logical variable. Unconventional uses of input and output are sometimes called "running programs backward." For example, we can use the same member relation to issue the query "is there a list XS that contains both 3 and 4 as members?"

S56b. ⟨*transcript* S45a⟩+≡ ◁S56a S57▷

```
?- member(3, XS), member(4, XS).
XS = [3, 4|_XS354]
yes
```

The resulting list contains an internal variable, _XS354, which indicates that the rest of the list is undetermined. In effect, Prolog says "yes, any list that begins with 3 and 4 will do." Such a result might surprise you, but it enables queries like member(3, XS) and member(4, XS) to interact with other queries or with subgoals that may determine _XS354. Sharing a logical variable is a powerful form of communication, because information can flow in multiple directions.

To summarize, the logical interpretation of Prolog answers a query by finding a substitution that makes the query is provable. The logical interpretation doesn't say *what* substitution is found; in the example query member(X, [4, 3]), Prolog finds X = 4, but according to the logical interpretation, X = 3 is just as good. The next step in our analysis of Prolog's semantics is to make the logical interpretation precise.

Making the logical interpretation precise

The logical interpretation of Prolog can be formalized using a simple, elegant, *nondeterministic* proof system. The formalization involves substitutions, which are presented in Chapter 7 as a means of implementing ML type inference, and which we revisit here.

Definition D.1 A *substitution* θ is a function θ from terms to terms that preserves structure, which is to say it satisfies these two equations:

$$\theta(\text{APPLY}(f, t_1, \ldots, t_n)) = \text{APPLY}(f, \theta(t_1), \ldots, \theta(t_n))$$
$$\theta(\text{LITERAL}(n)) = \text{LITERAL}(n)$$

Also, a substitution has a finite domain: for all but finitely many X, $\theta(\text{VAR}(X)) = \text{VAR}(X)$.

Substitutions have the following properties, which you might like to confirm (Exercise 36, page S109):

- Any substitution can be written as $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$. For any X that is not one of the X_i, θ leaves X unchanged; otherwise $\theta(\text{VAR}(X_i)) = t_i$. The set $\{X_1, \ldots, X_n\}$ is the *domain* of θ. We sometimes say that θ *binds* X_i to t_i, or that X_i is *bound in* θ.

- If functions θ_1 and θ_2 are substitutions, the composition $\theta_2 \circ \theta_1$ is also a substitution.

Since a goal has the form of a term, a substitution θ can be applied to a goal. A similar law applies: $\theta(p(t_1, \ldots, t_n)) = p(\theta(t_1), \ldots, \theta(t_n))$. For example, if

$$g = \texttt{member(X, [Y|YS])} \quad \text{and} \quad \theta = \{\texttt{X} \mapsto \texttt{3},\ \texttt{YS} \mapsto \texttt{[4|ZS]}\},$$

then $\theta(g) = $ member(3, [Y,4|ZS]).

Substitutions answer queries. That is, a query in Prolog is not simply satisfied—its satisfaction produces a substitution. Given query gs, the interpreter finds a substitution θ that makes $\theta(gs)$ provable. Examples are found throughout the chapter; the substitution is printed right after the query.

S57. ⟨*transcript* S45a⟩+≡ ◁S56b S58▷

```
-> [query].
?- britmap_coloring(Atl, En, Ie, NI, Sc, Wa).
Atl = yellow
En = blue
Ie = blue
NI = red
Sc = red
Wa = red
yes
```

Using substitutions, we can formalize Prolog's notion of query. To say "goal g is satisfiable using database D and substitution θ," we write the judgment $D \vdash \theta g$. In general, a query has more than one goal, so the general form of the judgment is

$$D \vdash \theta g_1, \ldots, \theta g_n.$$

In the logical interpretation, the satisfaction of the different goals is independent; the only requirement is that the *same* substitution satisfy them all. Formally,

$$\frac{D \vdash \theta g_i, \quad 1 \le i \le n}{D \vdash \theta g_1, \ldots, \theta g_n}. \qquad \text{(LogicalQueries)}$$

A single goal is satisfied if it can be "made the same" as the left-hand side of some clause whose right-hand side we can prove. Here, a crucial fact comes into play. *The variables used in a clause are arbitrary, bearing no relationship to variables of the same name that may appear in a query or in a subgoal from another clause (or even another instance of the same clause).* In other words, a variable in a Prolog clause is like a formal parameter of a function in another language; just as different activations of a function can bind different values to the "same" formal parameter, different uses of a clause can substitute different terms for the "same" logical variable.

Here's a contrived example. Suppose we want to find out if the variable `XS` is a member of the list `[1|nil]`. Variable `XS` is a strange name for an integer, but the answer is yes, provided `XS = 1`.

S58. ⟨*transcript* S45a⟩+≡ ◁S57 S61▷

```
?- member(XS, [1|nil]).
XS = 1
yes
```

How do we prove that this query is satisfied? By appealing to one of the clauses in the database: `member(X, [X|XS])`. The `XS` in the clause *must* be independent of the `XS` in the query, because `XS` cannot be both `1` and `nil` at the same time.

In Impcore, μScheme, and other languages, this kind of independence is achieved by using an environment to keep track of the values of formal parameters; each time a function is called, the activation gets its own private environment. In Prolog, this kind of independence is achieved by *renaming* the variables of each clause; each time a clause is used in a proof, the use gets its own private renaming.

Definition D.2 A *renaming of variables* is a substitution θ_α which is one-to-one and which maps every variable to a (possibly identical) variable, not to an application or an integer.

When considered as a function from terms to terms, a renaming of variables has an inverse function, which is also a substitution; we write that substitution θ_α^{-1}.

Using substitutions and renamings, Figure D.3 on page S59 presents a precise, inductive definition of the semantics of Prolog according to the logical interpretation. Judgment form $D \vdash \theta(gs)$ says that when substitution θ is applied to the list of goals gs, the conjunction of the goals is provable from clauses in database D. We say goals gs are *satisfied* by θ.

Formally, a list of goals is either an empty list $[\,]$ or a nonempty list $g :: gs$. A substitution is applied to a list by applying it to each element:

$$\theta([\,]) = [\,] \qquad\qquad \theta(g :: gs) = \theta(g) :: \theta(gs)$$

Judgment $D \vdash \theta(gs)$ is used with D and gs as inputs and θ as the output. There is one rule for each form of query. The empty list of goals is satisfied by

$$\boxed{D \vdash \theta(gs)} \qquad \textsc{EmptyQuery}\ \frac{}{D \vdash I([])}$$

$$\textsc{NonemptyQuery}\ \frac{\begin{array}{c} C \in D \qquad C = G \;\textbf{:-}\; H_1, \ldots, H_n \\ \theta_\alpha \text{ renames the free variables of } C \\ \theta(\theta_\alpha(G)) = \theta(g) \qquad D \vdash \theta([\theta_\alpha(H_1), \ldots, \theta_\alpha(H_n)]) \\ D \vdash \theta(gs) \end{array}}{D \vdash \theta(g :: gs)}$$

Figure D.3: The logical interpretation of Prolog

any database and the identity substitution I. A nonempty list is satisfied by tackling the goals one at a time, inductively; the key rule is NONEMPTYQUERY in Figure D.3. A single goal g is satisfied by θ if there is some clause C in the database such three conditions hold: C has head G; when variables in C are renamed, the renamed head $\theta_\alpha(G)$ unifies with g; and substitution θ also satisfies the (renamed) premises of C. And a nonempty list of goals $g :: gs$ is satisfied by a substitution θ if θ satisfies every goal in the list.

The NONEMPTYQUERY rule is wildly nondeterministic. There are three sources of nondeterminism, of which only one makes a real difference to the answer.

- The renaming θ_α can map the free variables of C to any set of variables that don't appear anywhere else. This nondeterminism makes no real difference to the answer; it affects θ only up to renaming.[4]

- The substitution θ must simultaneously satisfy three criteria: it must unify $\theta_\alpha(G)$ and g; it must satisfy the remaining goals gs, and it must satisfy the (renamed) premises $H_1, \ldots, H_n$. Even these three criteria don't determine θ completely; in Prolog, we expect to get a *most general* substitution satisfies these criteria (sidebar, page S60).

  This nondeterminism looks challenging, but in practice each of the criteria above corresponds to a subproblem, and it is not difficult to design an algorithm that computes a most general θ as the composition of lesser substitutions that solve each subproblem. The idea is exactly the same idea used to solve conjunctions in the constraint solver. And as in the constraint solver, changing the order in which the subproblems are solved may affect the answer, but only up to renaming.

- Clause C may be any clause in the database, or more precisely, it may be any clause whose head unifies with g. Unlike the other two forms of nondeterminism, this one really matters: which C is chosen makes a big difference to the answer θ.

In the logical interpretation of Prolog, a query gs is satisfied if there *exists* a derivation of $D \vdash \theta(gs)$. But unless D has only very boring inference rules, the number of potential derivations is unbounded, and the real questions are whether Prolog can *find* a derivation, and if so, *which* ones does it find? To answer these questions, we turn to the procedural interpretation.

[4] Two substitutions θ and θ' are *equivalent up to renaming* if there exists a renaming θ_β such that $\theta = \theta_\beta \circ \theta'$ (and therefore also $\theta' = \theta_\beta^{-1} \circ \theta$).

Unification, most general substitutions, and the occurs check

Definition D.3 A substitution θ_1 is *more general* than a substitution θ_2 if there exists a θ_3 such that $\theta_2 = \theta_3 \circ \theta_1$. That is, we can make θ_2 by composing something else with θ_1.

The more general a substitution is, the fewer things it changes.

Definition D.4 *Unification* is the standard algorithm for solving equality constraints. Given constraint $g_1 \sim g_2$, unification finds a substitution θ such that $\theta(g_1) = \theta(g_2)$. Furthermore, unification finds a θ that is a *most general* substitution satisfying this equation. Substitution θ is most general if for any θ' such that $\theta'(g_1) = \theta'(g_2)$, there is a substitution θ'' such that $\theta' = \theta'' \circ \theta$. In the examples below, I don't verify that the substitutions are most general substitutions.

Here are examples of unification problems and their solutions:

1. $g_1 = \texttt{member(3, [3|nil])}$
   $g_2 = \texttt{member(X, [X|XS])}$
   $\theta = \{\texttt{X} \mapsto \texttt{3},\ \texttt{XS} \mapsto \texttt{nil}\}$

2. $g_1 = \texttt{member(Y, [3|nil])}$
   $g_2 = \texttt{member(X, [X|XS])}$
   $\theta = \{\texttt{Y} \mapsto \texttt{3},\ \texttt{X} \mapsto \texttt{3},\ \texttt{XS} \mapsto \texttt{nil}\}$

3. $g_1 = \texttt{member(3, [4|nil])}$
   $g_2 = \texttt{member(X, [X|XS])}$

   do not unify, since no substitution can map `X` to both `3` and `4`.

4. $g_1 = \texttt{length([3|nil], N)}$
   $g_2 = \texttt{member(X, [X|XS])}$

   do not unify, since no substitution can make `length` equal `member`.

5. $g_1 = \texttt{member(X, [X|XS])}$
   $g_2 = \texttt{member(Y, cons(mkTree(Y, nil, nil), M))}$

   do not unify. Since the `X` in g_1 and the `Y` in g_2 must be replaced by the same term, say t, we end up with goals

   $$\begin{aligned} \theta(g_1) &= \texttt{member(}t\texttt{, [}t\texttt{|XS])} \\ \theta(g_2) &= \texttt{member(}t\texttt{, [mkTree(}t\texttt{, nil, nil)|XS])} \end{aligned}$$

   which cannot be unified: No substitution can make t equal to term `mkTree(`t`, nil, nil)`, because no matter what term is substituted for t, the number of appearances of `mkTree` will differ.

Example 5 illustrates a tricky aspect of implementing Prolog. Even if t is a logical variable, it does not unify with the term `mkTree(`t`, nil, nil)`. It is natural to try to unify a variable `X` with a term `t` using the substitution $\{\texttt{X} \mapsto \texttt{t}\}$, but this substitution works only if `X` does not occur in `t`. Unification of a variable with a term therefore requires an *occurs check*, which although expensive is an essential part of the semantics.

The procedural interpretation

Logic may be nondeterministic, but a logic program runs on a deterministic machine. The machine takes deterministic actions, like choosing a clause or trying to unify a goal with the clause's head. The procedural interpretation of Prolog says what actions are taken in what order. In particular, it tells us how the interpreter searches for clauses and how the interpreter computes and composes substitutions. Informally, the procedural interpretation of Prolog is just this: given database D and query gs, Prolog uses *depth-first search* to try to find a substitution θ and derivation of $D \vdash \theta(gs)$ using the rules in Figure D.3 on page S59. The search considers each $C \in D$ in the order in which the C's appear.

Depth-first search is simple in concept, but there are many details. To use Prolog effectively, you must understand how search works. You should know enough to estimate how your Prolog programs will perform, and you must know enough to avoid sending the search algorithm into an infinite loop. And to be at your must effective, you must know how to use the "cut" (Section D.8.3, page S91) to control the scope of Prolog's depth-first search.

Because the cut is a control operator, a formal semantics of the procedural interpretation is most easily expressed using a small-step semantics with an explicit evaluation context, like the one in Chapter 3. But such a semantics is unlikely to convey much understanding. If we omit the cut, then writing a big-step semantics is not so difficult, but it's best if you work it out for yourself (Exercise 37, page S110). Here, the procedural interpretation is presented informally, with examples. And because it involves so many details, it is presented in stages. The first stage explains how Prolog searches for clauses, without involving substitutions. The second stage explains how the search for clauses may *backtrack*, again without involving substitutions. The final stage adds substitutions.

Simple search for a matching clause

Given database $D = C_1, \ldots, C_n$ and query g, we wish to know whether g is satisfied, i.e. $D \vdash g$. To explain search without having to worry about substitutions, I assume that all clauses and goals are *ground*, that is, they have no variables. I also simplify the explanation by limiting my query to a single goal g. The simple search algorithm works in three steps:

1. Examine the clauses C_i *in the order in which they appear in* D. If no clause exists whose left-hand side is g, g is unsatisfied.

2. Otherwise, take the *first* clause whose left-hand side is g. Say that clause is g `:-` $H_1, \ldots, H_m$. Now recursively try to satisfy subgoals $H_1, \ldots, H_m$, in that order, using the same simple search algorithm.

3. If each H_j is satisfied, g is satisfied; if any H_j is unsatisfied, so is g.

You might feel uneasy that only the first clause is used in step 2, but this interpretation, although oversimplified, does explain the behavior of some variable-free programs. Here's an example:

S61. ⟨*transcript* S45a⟩+≡ ◁S58 S62▷

```
-> [rule].
-> imokay :- youreokay, hesokay.    /* clause C_1 */
-> youreokay :- theyreokay.         /* clause C_2 */
-> hesokay.                         /* clause C_3 */
-> theyreokay.                      /* clause C_4 */
-> [query].
```

```
?- imokay.
yes
```

The successful outcome is explained by the simple search algorithm:

- The goal is `imokay`. The first matching clause is C_1. Step 2 of the algorithm recursively tries to satisfy new goals `youreokay` and `hesokay`, which are called *subgoals*.

- Subgoal `youreokay` comes first. Clause C_2 matches and spawns subgoal `theyreokay`.

- Step 2 recursively tries to satisfy the subgoal `theyreokay`. The subgoal is matched by clause C_4, which spawns no new subgoals. So `theyreokay` is satisfied, and therefore so is `youreokay`.

- The recursive call returns, and the earlier step 2 continues by trying to solve the next subgoal: `hesokay`. This subgoal is matched by C_3, which spawns no subgoals. So `hesokay` is satisfied, and therefore so is `imokay`.

In this example, the search algorithm and the logical interpretation produce the same result. But some cases, the logical interpretation can answer a query when the simple search algorithm does not. To construct such a case, I add three clauses to our database:

S62. ⟨*transcript* S45a⟩+≡ ◁S61 S63a▷

```
?- [rule].
-> hesnotokay :- imnotokay.  /* clause $C_5$ */
-> shesokay :- hesnotokay.   /* clause $C_6$ */
-> shesokay :- theyreokay.   /* clause $C_7$ */
-> [query].
?- shesokay.
yes
```

According to the logical interpretation, `theyreokay` is a fact (clause C_4), and `shesokay` is provable from `theyreokay` by clause C_7. But the simple search algorithm does not prove `shesokay`. Rather, it tries to prove `shesokay` by applying C_6, which spawns subgoal `hesnotokay`, for which the algorithm tries to apply C_5, which spawns subgoal `imnotokay`, which cannot be proven.

What's wrong? More than one clause applies to the goal `shesokay`, and the first such clause doesn't lead to a solution. To fix this problem, we refine our view of the procedural interpretation by adding *backtracking*.

Backtracking search for matching clauses

As before, we have $D = C_1, \ldots, C_n$ and query g, and we wish to know whether g is satisfied. The backtracking search algorithm builds on the simple search algorithm, and the first two steps are identical:

1. Examine the clauses C_i *in the order in which they appear in* D. If no clause exists whose left-hand side is g, g is unsatisfied.

2. Otherwise, find a clause whose left-hand side is g. Say that clause is $C_i = g$ `:-` $H_1, \ldots, H_m$. Now recursively try to satisfy subgoals $H_1, \ldots, H_m$, in that order, using the same algorithm.

3. If each H_j is satisfied, g is satisfied; if any H_j is unsatisfied, don't give up—instead, repeat step 2 with the *next* clause in the database whose left-hand side is g, starting the search from clause C_{i+1}. Iteration continues until g is satisfied, or until there is no clause remaining whose left-hand side is g.

This backtracking algorithm is powerful enough to prove `shesokay`:

- Clause C_6 is the first clause that matches `shesokay`, and it spawns subgoal `hesnotokay`.
- Clause C_5 matches, and it spawns subgoal `imnotokay`.
- No clause matches subgoal `imnotokay`, so it is unsatisfied.
- The algorithm backtracks and continues trying to satisfy `hesnotokay`, starting from clause C_6. Clauses C_6 and C_7 don't match, so `hesnotokay` is unsatisfied.
- One level up in the recursion, the algorithm backtracks and continues trying to satisfy `shesokay`, starting from clause C_7. Clause C_7 matches and spawns subgoal `theyreokay`.
- Clause C_4 matches goal `theyreokay`, and there are no more subgoals. Goal `shesokay` is satisfied.

Backtracking gets us closer to the logical interpretation, but the two interpretations still don't agree. To show how they disagree, I add two more clauses:

S63a. ⟨*transcript* S45a⟩+≡ ◁S62 S64▷

```
?- [rule].
-> hesnotokay :- shesokay.   /* clause $C_8$ */
-> hesnotokay :- imokay.     /* clause $C_9$ */
```

Now my depth-first search goes into an infinite loop:

S63b. ⟨*bad transcript* S63b⟩≡

```
-> [query].
?- shesokay.
... never returns ...
```

In logic, if a conclusion can be inferred from some set of facts, it can still be inferred when new facts are added. Therefore, in the logical interpretation, `shesokay` is still provable after adding C_8 and C_9. But the backtracking search algorithm doesn't discover a proof; instead, it fails to terminate:

- Clause C_6 matches goal `shesokay` and spawns subgoal `hesnotokay`.
- Clause C_5 matches `hesnotokay`, spawning subgoal `imnotokay`, which still cannot be satisfied. The algorithm backtracks and continues trying to satisfy `hesnotokay`.
- Clause C_8 matches `hesnotokay` and spawns subgoal `shesokay`.
- Clause C_6 matches `shesokay` and spawns subgoal `hesnotokay`.
- And so on...

There may be a proof, but the algorithm doesn't find it, and even under the full procedural interpretation, the search algorithm loops forever. *The logical interpretation does not reflect the actual semantics of Prolog*. The procedural interpretation, which prescribes exactly how Prolog searches for clauses, is the accurate one.[5]

In logic, inference rules are unordered, or to put it another way, the order of clauses doesn't matter. But to Prolog's search algorithm, the order of clauses in

[5] There are other algorithms for logic programming, like *answer-set programming*, which are guaranteed to terminate. Such algorithms can even be applied to some Prolog programs, but they remain nonstandard interpretations of Prolog. Details are beyond the scope of this book.

the database is critically important. For example, if C_8 and C_9 are reversed, the search algorithm finds a proof of `shesokay`. While we might prefer a programming language based on pure logic, which always finds a solution when one exists, this is not how Prolog works.

Backtracking search for matching clauses, with variables

To get to the full algorithm that constitutes the procedural interpretation of Prolog, we have to say what happens to goals and clauses that include logical variables. In the general case, we are given a database D and a query $g_1, \ldots, g_k$. Each g_i may contain logical variables, and so may each clause. We want a θ such that $D \vdash \theta(g_1), \ldots, \theta(g_k)$. When $k = 0$, the empty query is trivially satisfied by the identity substitution. When $k = 1$, Prolog's search algorithm works as follows:

1. To satisfy a single goal g, examine the clauses C_i in the order in which they appear in D. If there is no clause with left-hand side G such that equality constraint $g \sim \theta_\alpha G$ can be solved, where θ_α is a renaming, g is unsatisfied.

2. Otherwise, find a clause $C_i = G$ `:-` $H_1, \ldots, H_m$, choose a renaming θ_α, and find a substitution θ such that $\theta(g) = \theta(\theta_\alpha(G))$. Letting $\theta' = \theta \circ \theta_\alpha$, recursively try to satisfy subgoals $\theta'(H_1), \ldots \theta'(H_m)$, in that order, using the general search algorithm that solves queries with multiple goals.

3. If each $\theta'(H_j)$ is satisfied, g is satisfied by substitution θ. If any H_j is unsatisfied, don't give up—instead, repeat step 2 with the next clause in the database whose left-hand side can be unified with g, starting the search from clause C_{i+1}. Iteration continues until g is satisfied, or until there is no clause remaining whose left-hand side is g.

When $k > 1$, that is when the query comprises multiple goals, such as might be produced from $\theta'(H_1), \ldots \theta'(H_m)$, Prolog composes substitutions:

$$\frac{D \vdash \theta_1(g_1) \qquad D \vdash \theta'(\theta_1(g_2)), \ldots, \theta'(\theta_1(g_k))}{D \vdash (\theta' \circ \theta)(g_1), \ldots, (\theta' \circ \theta)(g_k)} \qquad (\textsc{ProceduralQueries})$$

Informally, Prolog searches for a substitution θ_1 that satisfies goal g_1. If successful, it then tries to satisfy query $\theta_1(g_2), \ldots, \theta_1(g_k)$, an attempt which yields substitution θ'. The attempt to satisfy $g_1, \ldots, g_k$ has now succeeded, yielding the substitution $\theta' \circ \theta$. Or if you prefer, it solves goals $g_1, \ldots, g_k$ one at a time, accumulating substitution $\theta_k \circ \cdots \circ \theta_1$.

When Prolog solves queries with multiple goals in the presence of variables and substitutions, it needs a second kind of backtracking. To see why, let's return to an earlier example:

S64. ⟨*transcript* S45a⟩+≡ ◁S63a S69▷

```
-> [query].
?- member(X, [1, 2, 3, 4]), X > 2.
X = 3
yes
```

Goal g_1 is `member(X, [1, 2, 3, 4])`, and it is solved by substitution $\theta_1 = \{\texttt{X} \mapsto 1\}$. But when θ_1 is applied to goal g_2, which is `1 > 2`, the resulting subgoal is `1 > 2`, which is not solvable. But before giving up, Prolog asks if there is *another* substitution that solves g_1. Eventually it hits on $\{\texttt{X} \mapsto 3\}$, and `3 > 2` is solvable.

In the general case, here's what this part of the algorithm looks like. The problem is to solve query $g_1, \ldots, g_k$.

1. If $k = 0$, the query is solved by the identity substitution.

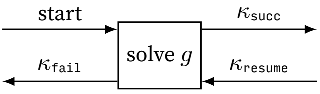
2. Otherwise, find substitution θ_1 that solves goal g_1. If there is no such θ_1, goal g_1 can't be solved.

3. Recursively find substitution θ' that solves $\theta_1(g_2), \ldots, \theta_1(g_k)$. If you find it, the entire query $g_1, \ldots, g_k$ is solved by substitution $\theta' \circ \theta_1$. If you don't find it, backtrack and ask if there is a *different* substitution θ_{1bis} that *also* solves g_1, and then try solving $\theta_{1bis}(g_2), \ldots, \theta_{1bis}(g_k)$. (There could be a different substitution θ_{1bis} because g_1 could unify with the head of a different clause.)

The search fails to solve the whole query only when *all* substitutions that solve g_1 have been exhausted.

The procedural interpretation illustrated using continuations

The full search algorithm that defines the procedural interpretation of Prolog can be hard to understand. Luckily there is a conceptual tool, the *Byrd box* (Byrd 1980), which not only makes it easier to understand how Prolog works, but which leads to a very simple implementation in continuation-passing style. You know the Byrd box already, from Section 2.10.2 (page 138), where it is used to solve Boolean formulas. In Prolog, the Byrd box is a "solver" for a single goal, with this structure:



The idea is simple:

1. We create a Byrd box for every goal g. The Byrd box searches for substitutions θ such that $D \vdash \theta(g)$.

2. There might be more than one such substitution, and we don't want to compute any more than necessary, so instead of simply having the Byrd box *return* a substitution, we pass it a *success continuation* $\kappa_{\mathtt{succ}}$. The continuation takes θ as a parameter.

3. Whether backtracking is needed depends on the goals that *follow* g; these are exactly the goals that $\kappa_{\mathtt{succ}}$ tries to satisfy. If they can't be satisfied, we go back to our original Byrd box and ask for another substitution. For this purpose, the Byrd box provides another continuation $\kappa_{\mathtt{resume}}$.

4. Finally, if the Byrd box fails, or if it simply runs out of substitutions, what do we do? We can't simply give up, because it's possible that backtracking might lead to another solution. So we pass the Byrd box a *failure continuation* $\kappa_{\mathtt{fail}}$, which it calls if it can't find a substitution, or if it has to backtrack.

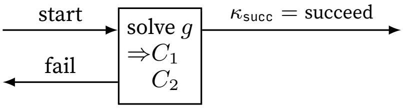
A Byrd box is implemented by function `solveOne` in ⟨*search* **⟦prototype⟧** S80⟩. Byrd boxes are illustrated below by three examples: two based on `member` and one based on map coloring.

The `member` relation has two proof rules:

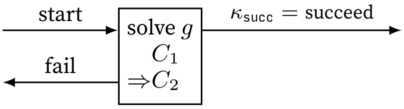
```
member(X, [X|XS]).                    /* C1 */
member(X, [Y|XS]) :- member(X, XS). /* C2 */
```

To answer the query g = `member(3, [4, 3]))`, the search algorithm takes these steps:

1. It creates a Byrd box that is prepared to consider clauses C_1 and C_2.[6]

   The goal does not unify with C_1's head, so the Byrd box changes state to look at C_2:

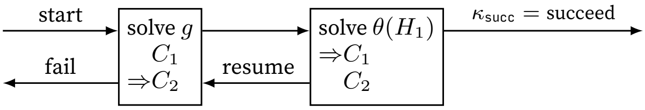
   

   The goal g does unify with C_2's head. Variables in C_2 are renamed so the head is `member(X1, [Y1|XS1])`, which unifies with g via substitution

   $$\theta = \{\texttt{X1} \mapsto \texttt{3},\ \texttt{Y1} \mapsto \texttt{4},\ \texttt{XS1} \mapsto [\texttt{3}]\}.$$

   The Byrd box spawns a new subgoal, $\theta(H_1)$, which is `member(3, [3])`.

2. The search algorithm now recursively tries to satisfy $\theta(H_1)$, which is term `member(3, [3])`. The algorithm creates a new Byrd box, and the new Byrd box gets the same success continuation as the current Byrd box. If the new goal fails, the search algorithm will continue looking for clauses after C_2.

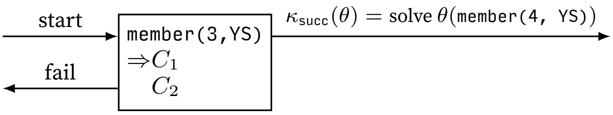
   Clause C_1 matches, via $\{\texttt{X2} \mapsto \texttt{3},\ \texttt{XS2} \mapsto \texttt{nil}\}$ (renaming `X` and `XS` in C_1 to `X2` and `XS2`). As C_1 has no subgoals, goal $\theta(H_1)$ is satisfied. Control passes to the success continuation, and query g is also satisfied.

Because query g has no variables, this example does not produce a substitution.

Our next example involves "running the program backward":

```
member(3, YS), member(4, YS).
```

1. To try to satisfy `member(3, YS)`, the search algorithm creates a Byrd box. If the attempt succeeds, the Byrd box's success continuation tries to solve `member(4, YS)`.

   
   start
fail
member(3,YS)
$\Rightarrow C_1$
C_2
$\kappa_{\mathsf{succ}}(\theta)$ = solve θ(member(4, YS))

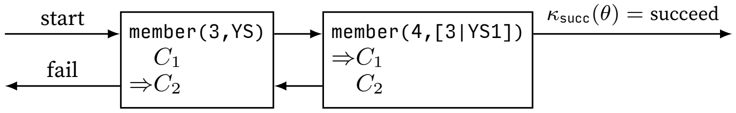
   Clause C_1 matches with equality constraint $\texttt{X1} \sim 3 \land \texttt{YS} \sim \texttt{[X1|XS1]}$, where `X` and `XS` in C_1 are renamed to `X1` and `XS1`. The constraint is solved by

   $$\theta_0 = \{\texttt{X1} \mapsto \texttt{3},\ \texttt{YS} \mapsto \texttt{[3|XS1]}\}$$

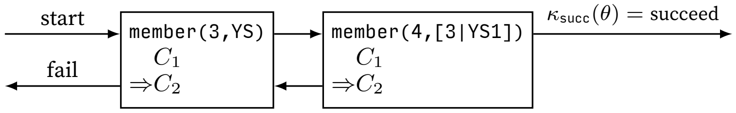
   We pass θ_0 to κ_{succ}.

[6] The semantics actually require that we consider all clauses, but these are the only clauses whose heads could possibly unify with query g.

2. The search algorithm creates a new Byrd box to solve $\theta_0(\texttt{member}(4, \texttt{YS}))$, which is `member(4, [3|YS1])`. If that fails, control will pass to the resume continuation, search will resume in the previous Byrd box at C_2.



Clause C_1 does not apply, because its head `member(X, [X|XS])` does not match the goal `member(4,[3|YS1])`. The current box moves to C_2.



This example illustrates a general property of Byrd boxes: at any one time, only the rightmost box is active.

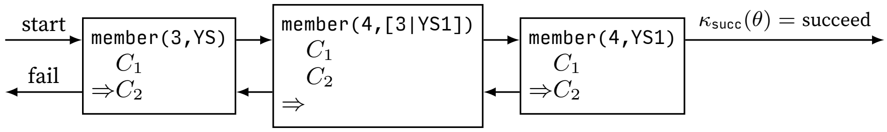
Continuing, the head of C_2 does match the goal: renaming `X`, `Y`, and `XS` in C_2 to `X2`, `Y2`, and `XS2` produces the equality constraint `member(X2, [Y2|XS2])` $\sim$ `member(4, [3|XS1])`. The constraint is satisfied by

$$\theta_0' = \{\texttt{X2} \mapsto \texttt{4},\ \texttt{Y2} \mapsto \texttt{3},\ \texttt{XS2} \mapsto \texttt{XS1}\}.$$

3. The search doesn't simply pass θ_0' to κ_{succ}; it first tackles the subgoal spawned by clause C_2, which, applying the substitution, is:

$$\theta_0'(\texttt{member(X2, XS2)}) = \texttt{member(4, YS1)}.$$

Again, the algorithm creates a new box.



Clause C_1 matches, yielding[7] $\theta_1' = \{\texttt{X3} \mapsto \texttt{4},\ \texttt{YS1} \mapsto \texttt{[4|YS3]}\}$.

4. Subgoal `member(4, [3|YS1])` (step 2) is now satisfied by substitution

$$\begin{aligned}\theta_1 &= \theta_1' \circ \theta_0' \\ &= \{\texttt{X3} \mapsto \texttt{4},\ \texttt{YS1} \mapsto \texttt{[4|YS3]},\ \texttt{X2} \mapsto \texttt{4},\ \texttt{Y2} \mapsto \texttt{3},\ \texttt{XS2} \mapsto \texttt{[4|YS3]}\}.\end{aligned}$$

5. The original goal is satisfied by

$$\theta_1 \circ \theta_1 = \{\texttt{X1} \mapsto \texttt{3},\ \texttt{X3} \mapsto \texttt{4},\ \texttt{YS1} \mapsto \texttt{[4|YS3]},\ \texttt{YS} \mapsto \texttt{[3,4|YS3]},\ \ldots\}.$$

Our third example of Prolog search uses the `britmap_coloring` query, which allows us to explore backtracking within right-hand sides while avoiding equality constraints, unification, and renaming of variables. The computation that solves the query `britmap_coloring(Atl, En, Ie, NI, Sc, Wa)` is long, so I show only the first dozen steps or so. Fortunately, only one clause matches this goal, but it spawns a lot of subgoals (ignoring the renaming of variables):

```
different(Atl, En), different(Atl, Ie), different(Atl, NI), ...
```

[7] From here, I don't explain each individual renaming of variables. Each time I need to rename a variable, I append the next higher integer to its original name.

The search algorithm follows these steps:

1. Goal `different(Atl, En)` unifies with the first `different` clause in the database: `different(yellow, blue)`. The result is the first substitution $\theta_1 = \{\texttt{Atl} \mapsto \texttt{yellow},\ \texttt{En} \mapsto \texttt{blue}\}$.
2. Goal $\theta_1(\texttt{different(Atl, Ie)}) = \texttt{different(yellow, Ie)}$ is satisfied by substitution $\theta_2 = \{\texttt{Ie} \mapsto \texttt{blue}\}$.
3. Goal $\theta_2(\theta_1(\texttt{different(Atl, NI)})) = \texttt{different(yellow, NI)}$ is satisfied by substitution $\theta_3 = \{\texttt{NI} \mapsto \texttt{blue}\}$.
4. Goal $\theta_3(\theta_2(\theta_1(\texttt{different(Atl, Sc)}))) = \texttt{different(yellow, Sc)}$ is satisfied by substitution $\theta_4 = \{\texttt{Sc} \mapsto \texttt{blue}\}$.
5. Goal $\theta_4(\theta_3(\theta_2(\theta_1(\texttt{different(Atl, Wa)})))) = \texttt{different(yellow, Wa)}$ is satisfied by substitution $\theta_5 = \{\texttt{Wa} \mapsto \texttt{blue}\}$.
6. Goal $\theta_5(\theta_4(\theta_3(\theta_2(\theta_1(\texttt{different(En, Sc)}))))) = \texttt{different(blue, blue)}$ cannot be satisfied.
7. Backtracking to the previous subgoal, goal `different(yellow, Wa)` is resatisfied, yielding $\theta_5' = \{\texttt{Wa} \mapsto \texttt{red}\}$.
8. Goal $\theta_5'(\theta_4(\theta_3(\theta_2(\theta_1(\texttt{different(En, Sc)}))))$ is still `different(blue, blue)` and still cannot be satisfied.
9. Backtracking, no more substitutions satisfy `different(yellow, Wa)`. The algorithm backtracks to the previous subgoal, `different(yellow, Sc)`, and it satisfies the subgoal with a new substitution $\theta_4' = \{\texttt{Sc} \mapsto \texttt{red}\}$.
10. Like step 5.
11. Like step 6, but this time the goal is $\theta_5(\theta_4'(\theta_3(\theta_2(\theta_1(\texttt{different(En, Sc)})))))$, which is `different(blue, red)`, and the goal is satisfied. Substitution θ_6 is the identity substitution, which I ignore.
12. Goal $\theta_5(\theta_4'(\theta_3(\theta_2(\theta_1(\texttt{different(En, Wa)}))))) = \texttt{different(blue, red)}$, and the goal is satisfied.
13. Goal $\theta_5(\theta_4'(\theta_3(\theta_2(\theta_1(\texttt{different(Ie, NI)}))))) = \texttt{different(blue, blue)}$, which cannot be satisfied.

More backtracking is needed, but finishing this computation is up to you (Exercise 8, page S100).

D.3.5 Primitive predicates

The primitive predicates of μProlog are `true`, `atom`, `print`, `not`, `is`, `<`, `>`, `=<`, and `>=`.

- *`true`:* Always succeeds, with the identity substitution, provided it is not given any arguments. Has no side effects.
- *`atom`:* Takes one argument, which is a term. If the term is an atom, `atom` succeeds. If the term is an application, a number, or a logical variable, `atom` fails.
- *`print`:* Takes any number of terms as arguments, prints each of them, and succeeds.

- `not`*:* Takes one argument, which is interpreted as a *goal* g. Prolog tries to satisfy g. If g is satisfiable, `not` fails; otherwise, `not` succeeds (with the identity substitution). Regrettably, the predicate `not` is not simple logical negation; to understand `not`, you have to understand the procedural interpretation (see Section D.8.3).

- `is`*:* Takes two arguments, the second of which *must* be a term that stands for an arithmetic expression. Such a term can be

  - A literal integer
  - A variable that is instantiated to an integer
  - $e_1 \oplus e_2$, where e_1 and e_2 are terms that stand for arithmetic expressions, and $\oplus$ is one of these operators: `+`, `*`, `-`, or `/`.

  To use `is` with any other term is a checked run-time error.

  The predicate `is` works as follows: it computes the value of the expression, then looks at the first argument. If the first argument is an integer, then `is` succeeds if and only if the first argument is equal to the value denoted by the second. If the first argument is a variable, then `is` succeeds and produces the substitution mapping that variable to value denoted by the second argument. If the first argument is neither an integer nor a variable, `is` fails.

  S69. ⟨*transcript* S45a⟩+≡ ◁S64 S71b▷

```
-> [query].
?- 12 is 10 + 2.
yes
?- X is 2 - 5.
X = -3
yes
?- X is 10 * 10, Y is (X + 1) / 2.
X = 100
Y = 50
yes
```

- `<`, `=<`, `>`, `>=`*:* The primitive comparisons take two arguments, both of which must be instantiated to integers. They succeed or fail according to the way the integers compare.

The restrictions on the arguments of numeric predicates prevent infinite backtracking. If the restrictions were lifted, we could present a goal like `X is Y + 10`. But this goal is satisfied by an infinite number of substitutions! For every integer m, there is an integer $n = m + 10$, and the substitution $\{\mathtt{X} \mapsto n,\ \mathtt{Y} \mapsto n\}$ satisfies the goal. Therefore there are an infinite number of ways to attack any goal that would *follow* `X is Y + 10`, and if the following goal were not satisfiable, the result would be an infinite loop. To avoid such loops, Prolog disallows logical variables on the right-hand side of `is`.

D.4 More small programming examples

D.4.1 Lists

Prolog supports programming idioms that are impossible in Scheme or ML. To explore these idioms, let's look at lists again. Both Prolog and ML build lists using `cons` and `nil` (or `'()`), and both support pattern matching.

As a first example, list membership can be written as a (recursive) μML function:

```
(define member? (x xs)
   (case xs
     ['()           #f]
     [(cons y ys)   (if (= x y) #t (member? x ys))]))
```

For comparison, membership can also be defined as a (recursive) predicate:

```
-> member(X, [X|XS]).
-> member(X, [Y|YS]) :- member(X, YS).
```

The nonessential differences conceal some underlying similarities:

- Both languages use pattern matching—the μML pattern `(cons y ys)` is the same as the Prolog pattern `[Y|YS]`.

- Both languages distinguish a *variable,* which may be *bound* in a pattern, from a nonvariable, which may only be matched in a pattern. In μML, the nonvariable is called a "value constructor"; in Prolog, the nonvariable is called a "functor."

- To distinguish variables from nonvariables, each language has a spelling convention—but they use opposite conventions. In Prolog, a name beginning with a capital letter refers to a variable, and a name beginning with a lower-case letter refers to a functor. In μML, it's the other way round: a name beginning with a capital letter refers to a value constructor, and a name beginning with a lower-case letter refers to a variable. (Muddying the waters is the name `cons`; for consistency with Scheme, `cons` is considered a value constructor in μML as well as in μProlog.)

Prolog was the first widely used language to provide pattern matching, and Prolog's pattern matching is strictly more expressive than the pattern matching found in functional languages like Erlang, Haskell, and ML. In the functional languages, only one of the two terms to be matched may contain variables, and no variable may appear more than once. These restrictions enable a pattern match in a functional language to be compiled into machine code that is significantly more efficient than the code for Prolog's unification.

The essential differences are more interesting:

- Prolog doesn't have an equality predicate! Equality is tested by using the same variable multiple times in a rule—a variable is always equal to itself.

```
-> member (X, [X|XS]).            /* repeats X; correct idiom */
-> member (X, [Y|YS]) :- X = Y.   /* wrong! there is no = */
```

- μProlog doesn't use conditionals. Instead, for each condition under which a predicate can be shown to hold, we write a rule.

- Because nothing is a member of the empty list, there is no rule for membership of an empty list! This example highlights a big difference between functional programming and logic programming. If you write a function, that function has to return a value, even if the value represents falsehood. In logic programming, you write down only things that are true—or rather, that can be proved. If Prolog can't prove a fact or can't satisfy a predicate, it just assumes that the fact is false or the predicate is unsatisfiable. This assumption is called the *closed-world assumption.* The closed-world assumption can mislead you into thinking something isn't true when it really is. That's because Prolog doesn't deal in truth or falsehood; it deals in provability. If your inference rules aren't good enough to prove a fact, then to Prolog, that fact is as good as dead.

Now let's investigate some logic-programming idioms. At first I present logical predicates not only in Prolog but also in informal English and in inference rules; later I leave informal English and inference rules to you. As you read, I encourage you to think primarily about the logical interpretation of Prolog; where you need to be aware of the procedural interpretation, I point it out.

Our first example predicate, `snocced(`XS`, `X`, `YS`)`, holds if YS is the list obtained by adding X to the *end* of XS. Why "`snocced`"? To add an element to the beginning of a list, we use `cons`. And to add an element to the end of a list, we traditionally define `snoc`, which is `cons` spelled backward. The past participle of `snoc` is `snocced`.

S71a. ⟨*example queries of* `snocced` S71a⟩≡ (S71b) S71c ▷

```
?- snocced([3], 4, [3,4]).
yes
```

A claim of `snocced` can be justified by the following rules:

- The list obtained by adding X to the end of the empty list is $[X]$, a list of one element.

- The list obtained by adding X to the end of $[Y\,|\,YS]$ is $[Y|ZS]$, where ZS is the list obtained by of adding X to the end of YS.

In the notation of mathematical logic, these rules are written as follows:

$$\frac{}{\texttt{snocced}([\,], X, [X])} \qquad \frac{\texttt{snocced}(YS, X, ZS)}{\texttt{snocced}([Y\,|\,YS], X, [Y|ZS])}.$$

And in Prolog, the rules are written as follows:

S71b. ⟨*transcript* S45a⟩+≡ ◁ S69 S72a ▷

```
?- [rule].
-> snocced([], X, [X]).
-> snocced([Y|YS], X, [Y|ZS]) :- snocced(YS, X, ZS).
-> [query].
⟨example queries of snocced S71a⟩
```

To simulate a `snoc` function, we write queries of the form `snocced`(XS, X, YS), where X and XS are terms and YS is a logical variable:

S71c. ⟨*example queries of* `snocced` S71a⟩+≡ (S71b) ◁ S71a S71d ▷

```
?- snocced([3], 4, YS).
YS = [3, 4]
yes
```

But the `snocced` predicate can be used for other queries. For example, what list XS, when 4 is added to the end, produces the list $[3, 4]$?

S71d. ⟨*example queries of* `snocced` S71a⟩+≡ (S71b) ◁ S71c

```
-> snocced(XS, 4, [3, 4]).
XS = [3]
yes
```

Next let's look at list reversal. Predicate `reversed(`XS`, `YS`)` holds when YS is the reverse of XS. Here are a couple of rules:

S72a. ⟨*transcript* S45a⟩+≡ ◁S71b S72b▷

```
?- [rule].
-> reversed([], []).
-> reversed([X|XS], YS) :- reversed(XS, ZS), snocced(ZS, X, YS).
```

The code can be run in both directions:

S72b. ⟨*transcript* S45a⟩+≡ ◁S72a S72d▷

```
-> [query].
?- reversed([1, 2], XS).
XS = [2, 1]
yes
?- reversed(XS, [1, 2]).
XS = [2, 1]
yes
```

Another popular example is list append; in Prolog it works out especially neatly. Predicate `appended(`XS`, `YS`, `ZS`)` holds if ZS is the result of appending YS to XS, as in

S72c. ⟨*example queries of* `appended` S72c⟩≡ (S72d) S72f▷

```
?- appended([3, 4], [5], [3, 4, 5]).
yes
```

In the forward direction, `appended` is used to find ZS given XS and YS; in the backward direction, `appended` splits ZS into two pieces—*in every possible way*.

The rules that define the predicate `appended` are almost identical to what you would see in a clausal definition of function `append` in μML:

S72d. ⟨*transcript* S45a⟩+≡ ◁S72b S73b▷

```
?- [rule].
-> appended([], YS, YS).
-> appended([X|XS], YS, [X|ZS]) :- appended(XS, YS, ZS).
-> [query].
⟨example queries of appended S72c⟩
```

The μML function has the same structure:

S72e. ⟨*μML clausal definition of* `append` S72e⟩≡

```
(define* [(append '()         ys)  ys]
         [(append (cons x xs) ys)  (cons x (append xs ys))])
```

Back to Prolog, here are a forward and a backward example of `appended`.

S72f. ⟨*example queries of* `appended` S72c⟩+≡ (S72d) ◁S72c S72g▷

```
?- appended([3, 4], [5, 6], ZS).
ZS = [3, 4, 5, 6]
yes
?- appended(XS, YS, [5, 6, 7]).
XS = []
YS = [5, 6, 7]
yes
```

Let's split a list into nonempty sublists. A list is nonempty if formed with `cons`.

S72g. ⟨*example queries of* `appended` S72c⟩+≡ (S72d) ◁S72f S73a▷

```
?- [rule].
-> nonempty([X|XS]).
-> [query].
```

Now I split $[5, 6, 7]$ into two nonempty lists. The singleton list $[99]$ cannot be so split:

S73a. ⟨*example queries of* appended S72c⟩+≡ (S72d) ◁ S72g

```
?- appended(XS, YS, [5, 6, 7]), nonempty(XS), nonempty(YS).
XS = [5]
YS = [6, 7]
yes
?- appended(XS, YS, [99]), nonempty(XS), nonempty(YS).
no
```

As another example, appended can be used in the backward direction to define list membership:

S73b. ⟨*transcript* S45a⟩+≡ ◁ S72d S73c ▷

```
?- [rule].
-> member_variant(X, XS) :- appended(YS, [X|ZS], XS).
```

Only one clause is needed! Predicate member_variant means the same as member, whose definition uses two clauses.

Our last list example uses predicate member to define the equivalent of find from μScheme. We represent an association list as a list whose elements have the form pair(*key*, *attribute*), e.g.,

```
[pair(chile, santiago), pair(peru, lima), pair(brazil, brasilia)]
```

The predicate found(K, A, L) holds when association list L maps attribute A to key K. The found predicate can be defined in a single clause:

S73c. ⟨*transcript* S45a⟩+≡ ◁ S73b S73d ▷

```
-> found(K, A, L) :- member(pair(K, A), L).
```

This example also shows how to use a predicate to name a term, which is a bit like a LET binding; in this case, we associate the name capitals with the list above:

S73d. ⟨*transcript* S45a⟩+≡ ◁ S73c S73e ▷

```
-> capitals([pair(chile, santiago), pair(peru, lima), pair(brazil, brasilia)]).
```

To query the list of capitals, we begin the query with capitals(CS), then use CS in the remaining goals.

S73e. ⟨*transcript* S45a⟩+≡ ◁ S73d S74a ▷

```
-> [query].
?- capitals(CS), found(peru, CapitalOfPeru, CS).
CS = [pair(chile, santiago), pair(peru, lima), pair(brazil, brasilia)]
CapitalOfPeru = lima
yes
```

D.4.2 Arithmetic

Arithmetic predicates, as you might suspect from the restrictions on the primitive is predicate, are used primarily to code functions. A function that takes k parameters can be turned into a predicate of $k + 1$ values; the final place of the predicate typically stands for the result of the function you originally had in mind. I present two examples: power and factorial.

A function to raise a number to an integer power takes two arguments, so when expressed as a predicate, it becomes a three-place predicate. The predicate power(X, N, Z) holds when $Z = X^N$. The rules for power rely on two properties of exponentiation, which amount to a definition that is inductive in N:

- $X^0 = 1$, for any X.
- $X^N = X \cdot X^{N-1}$, for any N and X.

Each property can be expressed as a Prolog clause:

S74a. ⟨*transcript* S45a⟩+≡ ◁S73e S74b▷

```
?- [rule].
-> power(X, 0, 1).
-> power(X, N, Z) :- N > 0, N1 is N - 1, power(X, N1, Z1), Z is Z1 * X.
```

The subgoal `N > 0` prevents infinite recursion during backtracking.

We can use `power` in the forward direction:

S74b. ⟨*transcript* S45a⟩+≡ ◁S74a S74c▷

```
-> [query].
?- power(3, 5, Z).
Z = 243
yes
?- power(5, 3, Z).
Z = 125
yes
```

In logic, nothing prevents us from asking about the `power` predicate in other ways, but the results don't make anyone happy:

S74c. ⟨*transcript* S45a⟩+≡ ◁S74b S74e▷

```
?- power(3, N, 27).
Run-time error: Used comparison > on non-integer term
```

What happened? To understand this failure, we must appeal to the search algorithm that defines the procedural interpretation of Prolog. The second `power` clause matches, yielding subgoals `N > 0`, `N1 is N - 1`, and so on. But the predefined predicates `>` and `is N - 1` may be used only when `N` is instantiated to an integer. Because `N` is a logical variable, we get a checked run-time error.

Another consequence of the procedural interpretation (and of the definition of `is`) is that to make `power` work, its second clause must be written in the right way. Here is a wrong way to do it:

S74d. ⟨*bad version of* `power` S74d⟩≡

```
-> power(X, N, Z) :- N > 0, N1 is N - 1, Z is Z1 * X, power(X, N1, Z1).
```

This version is bad for reasons I ask you to figure out for yourself (Exercise 18, page S103).

Our other example definition, of a factorial predicate, looks a lot like `power`. It too is based on an inductive definition of a function.

S74e. ⟨*transcript* S45a⟩+≡ ◁S74c S74f▷

```
?- [rule].
-> fac(0, 1).
-> fac(N, R) :- N1 is N - 1, fac(N1, R1), R is N * R1.
```

Like `power`, `fac` runs only in the forward direction, and it works only because the subgoals in the second clause are written in the right order. And `fac` exhibits another subtle problem, which you can investigate in Exercise 20 (page S103).

D.4.3 Sorting

It is a theorem of arithmetic that any list of integers can be sorted. The theorem can be summarized in one clause:

S74f. ⟨*transcript* S45a⟩+≡ ◁S74e S75a▷

```
?- [rule].
-> sorted(XS, YS) :- permutation(XS, YS), ordered(YS).
```

Given definitions of permutation and ordered, sorted can be used to sort—but not very quickly.

S75a. ⟨*transcript* S45a⟩+≡ ◁ S74f S75b ▷

```
-> ordered([]).
-> ordered([N]).
-> ordered([N, M|NS]) :- N =< M, ordered([M|NS]).
-> permutation([], []).
-> permutation(XS, [Y|YS]) :-
       appended(WS, [Y|US], XS), appended(WS, US, ZS), permutation(ZS, YS).
```

The definition of ordered is simple. In permutation, I generate permutations by running appended in the backward direction, which splits list XS in all possible ways. The clauses say that:

- [] is a permutation of [].
- [$Y \mid YS$] is a permutation of XS if Y is an element of XS and YS is a permutation of the remaining elements. That is, [$Y \mid YS$] is a permutation of XS if XS can be split into two parts, WS and [$Y \mid US$], such that YS is a permutation of ZS, where ZS is the list we get by appending US to WS.

A query on sorted tries all permutations of its argument—as many as $n!$ for a list of length n—until it finds a sorted one.

S75b. ⟨*transcript* S45a⟩+≡ ◁ S75a S75c ▷

```
-> [query].
?- sorted([4, 2, 3], NS).
NS = [2, 3, 4]
yes
```

What an awful sorting algorithm! To define a better one, we once again turn a function into a predicate. As an example, here is Quicksort.

The key to Quicksort is the predicate partitioned(Pivot, XS, YS, ZS), which holds when YS and ZS form a partition of XS in which YS contains the elements less than or equal to Pivot and ZS contains the elements greater than Pivot. When we use partitioned in the forward direction, we supply a Pivot and XS that are instantiated to a specific value and list, respectively; but YS and ZS are logical variables. Satisfying a partitioned goal binds resulting lists to both YS and ZS.

S75c. ⟨*transcript* S45a⟩+≡ ◁ S75b S76a ▷

```
?- [rule].
-> partitioned(Pivot, [A|XS], [A|YS], ZS) :- A =< Pivot, partitioned(Pivot, XS, YS, ZS).
-> partitioned(Pivot, [A|XS], YS, [A|ZS]) :- Pivot < A,  partitioned(Pivot, XS, YS, ZS).
-> partitioned(Pivot, [], [], []).
-> quicksorted([], []).
-> quicksorted([X|XS], Sorted) :-
     partitioned(X, XS, Lows, Highs),
     quicksorted(Lows, Lows1), quicksorted(Highs, Highs1),
     appended(Lows1, [X|Highs1], Sorted).
```

One advantage of programming with logic is that important preconditions, invariants, and postconditions can be expressed as named predicates. When you understand what "sorted" and "partitioned" mean, the quicksorted clauses express the algorithm clearly.

Another advantage of logic programming is that unlike functions, relations can easily be code to "return" multiple results. In other languages, like C, Scheme, ML, and Smalltalk, a partition function has to return some sort of pair, record, or object containing the two halves of the partition. In Prolog, we could do the same—writing something like partitioned(X, XS, pair(Lows, Highs)), for example—but it is more idiomatic simply to make a place in the predicate for each result.

We just think of `partitioned` as a 4-place predicate that expects two inputs and produces two outputs. In Prolog, using a single predicate to compute multiple values comes naturally.

Here is an example use of `quicksorted`, in the forward direction:

S76a. ⟨*transcript* S45a⟩+≡ ◁S75c S76b▷

```
-> [query].
?- quicksorted([8, 2, 3, 7, 1], S).
S = [1, 2, 3, 7, 8]
yes
```

To explain why `quicksorted` can't be used in the backward direction is the task of Exercise 19 (page S103).

D.4.4 Difference lists

In the examples above, data is represented by *ground terms*. A ground term is one with no logical variables, or to define it inductively, a ground term is one of the following:

- An integer
- A nullary functor
- A functor applied to one or more ground terms

This is a fine way to represent data—it is essentially the same way data is represented in ML—but it doesn't take advantage of the full power of logic programming. It is also possible to represent data in a way that involves logical variables. An example that is both interesting and widely used is the *difference list*.

A difference list represents a list XS as the difference between two others lists YS and ZS. More precisely, a difference list is a term of the form `diff(`YS`, `ZS`)`, where ZS is a logical variable YS is a sequence of elements cons'ed onto ZS. For example, the term

```
diff([3,4|ZS], ZS)
```

represents the list containing the two elements `3` and `4`, i.e. the ordinary list `[3, 4]`. As another example, the term `diff(ZS, ZS)` represents the empty list. The interesting property of the difference list is that it can be refined by substituting for `ZS`.

A difference list can easily be transformed to an ordinary list, and vice versa. The predicate `canonical(`D`, `XS`)` is true if XS is the canonical, ordinary representation of the list represented by D.

S76b. ⟨*transcript* S45a⟩+≡ ◁S76a S77a▷

```
?- [rule].
-> canonical(diff(ZS, ZS), []).
-> canonical(diff([X|YS], ZS), [X|XS]) :- canonical(diff(YS, ZS), XS).
```

The definition is based on these facts:

- The difference between any list ZS and itself, `diff(`ZS`, `ZS`)`, represents the empty list.
- If the difference between `YS` and `ZS` is `XS`, then the difference between `[X|YS]` and `ZS` is `[X|XS]`.

The rules are easier to motivate if I write `diff` using a $-$ sign and `cons` using a $+$ sign:

$$\frac{}{ZS - ZS = [\,]} \qquad \frac{YS - ZS = XS}{(X + YS) - ZS = X + XS}.$$

Substitute for XS in the conclusion of the second rule, and you get the equation

$$(X + YS) - ZS = X + (YS - ZS).$$

The `canonical` predicate can transform lists in either direction.

S77a. ⟨*transcript* S45a⟩+≡ ◁ S76b S77b ▷

```
-> [query].
?- canonical(diff([3, 4|YS], YS), XS).
YS = _ZS6748
XS = [3, 4]
yes
?- canonical(D, [3, 4]).
D = diff([3, 4|_ZS6990], _ZS6990)
yes
```

One of the neat things about difference lists is that you can append them without any induction or recursion:

S77b. ⟨*transcript* S45a⟩+≡ ◁ S77a S77c ▷

```
?- [rule].
-> diffappended(diff(XS, YS), diff(YS, ZS), diff(XS, ZS)).
```

To get some intuition for this rule, look at this algebraic law:

$$(XS - YS) + (YS - ZS) = (XS - ZS).$$

We can use `diffappended` in the forward direction to append `[1, 2]` to `[3, 4]`:

S77c. ⟨*transcript* S45a⟩+≡ ◁ S77b S83a ▷

```
-> [query].
?- diffappended(diff([1, 2|YS], YS), diff([3, 4|ZS], ZS), D).
YS = [3, 4|_ZS7075]
ZS = _ZS7075
D = diff([1, 2, 3, 4|_ZS7075], _ZS7075)
yes
```

In this example, Prolog needs to make the goal equal to the head of the single clause for `diffappended`. Once the variables in the clause are renamed, the interpreter must unify these terms:

```
diffappended(diff([1,2|YS], YS),  diff([3,4|ZS], ZS),  D)
diffappended(diff(XS1,      YS1), diff(YS1,      ZS1), diff(XS1, ZS1))
```

These terms are made equal by the substitution

$$\begin{aligned}\theta = \{\ & \texttt{ZS} && \mapsto \texttt{ZS1}\\ ,\ & \texttt{YS} && \mapsto \texttt{[3,4|ZS1]}\\ ,\ & \texttt{YS1} && \mapsto \texttt{[3,4|ZS1]}\\ ,\ & \texttt{XS1} && \mapsto \texttt{[1,2,3,4|ZS1]}\\ ,\ & \texttt{D} && \mapsto \texttt{diff([1,2,3,4|ZS1], ZS1)}\\ \}. & && \end{aligned}$$

In the Prolog interpreter, renaming produces `_ZS7075` instead of `ZS1`, and with that change, substitution θ gives the answer.

Some other predicates on difference lists can also be coded without induction or recursion, and some other predicates, like `quicksorted`, are simpler when using difference lists (Exercise 17, page S103).

D.5 Implementation

The implementation of μProlog differs most obviously from my other implementations in two ways:

- There are no “values” as distinct from “abstract syntax”; terms do duty as both.
- There is no “evaluation.”[8] Instead, there are queries.

The main features of the implementation are the database, substitution, unification, and the backtracking query engine. They are presented below.

D.5.1 *The database of clauses*

I treat the database of clauses as an abstraction, which I characterize by its operations.

- We can add a clause to the database.
- Given a goal, we can search for clauses whose conclusions may match that goal.

Searching for potentially matching clauses is an important part of Prolog, and it can be worth choosing a representation of the database to make this operation fast (Exercise 43). If we do so, we have to preserve the *order* of the clauses in the database.

My representation is a list. As a result, I treat *every* clause as a potential match.

S78a. ⟨*μProlog’s database of clauses* S78a⟩≡ (S593b)

```
type database
emptyDatabase    : database
addClause        : clause * database -> database
potentialMatches : goal * database -> clause list
```

```
type database = clause list
val emptyDatabase = []
fun addClause (r, rs) = rs @ [r] (* must maintain order *)
fun potentialMatches (_, rs) = rs
```

D.5.2 *Substitution, free variables, and unification*

As part of type inference, Chapter 7 develops a representation of substitutions, as well as utility functions that apply substitutions to types. Prolog uses the same representation, but instead of substituting types for type variables, Prolog substitutes terms for logical variables. The code, which closely resembles the code in Chapter 7, is in Section V.4. Substitutions are discovered by solving equality constraints, which are defined here:

S78b. ⟨*substitution and unification* S78b⟩≡ (S593a) S79d ▷

```
type subst
idsubst : subst
|-->    : name * term -> subst
varsubst    : subst -> (name   -> term)
termsubst   : subst -> (term   -> term)
goalsubst   : subst -> (goal   -> goal)
clausesubst : subst -> (clause -> clause)
type con
consubst    : subst -> (con -> con)
```

```
datatype con = ~  of term * term
             | /\ of con  * con
             | TRIVIAL
infix 4 ~
infix 3 /\
```

⟨*free variables (terms/goals/clauses)* S79a⟩
⟨*substitutions for μProlog* S596b⟩

[8] Well, hardly any. The primitive is does a tiny amount of evaluation.

Free variables

The function `termFreevars` computes the free variables of a term. For readability, those free variables are ordered by their first appearance in the term, when reading from left to right. Similar functions compute the free variables of goals and clauses.

S79a. ⟨*free variables (terms/goals/clauses)* S79a⟩≡ (S78b)

```
termFreevars   : term   -> name set
goalFreevars   : goal   -> name set
clauseFreevars : clause -> name set
```

```
fun termFreevars t =
  let fun f (LITERAL _, xs) = xs
        | f (VAR x,     xs) = insert (x, xs)
        | f (APPLY(_, args), xs) = foldl f xs args
  in  reverse (f (t, []))
  end
fun goalFreevars goal = termFreevars (APPLY goal)
fun union' (s1, s2) = s1 @ diff (s2, s1)   (* preserves order *)
fun clauseFreevars (c :- ps) =
  foldl (fn (p, f) => union' (goalFreevars p, f)) (goalFreevars c) ps
```

Renaming variables in clauses: "Freshening"

Every time a clause is used, its variables are renamed. To rename a variable, I put an underscore in front of its name and a unique integer after it. Because the parser in Section V.7 does not accept variables whose names begin with an underscore, these names cannot possibly conflict with the names of variables that appear in source code.

S79b. ⟨*renaming μProlog variables* S79b⟩≡ (S593a) S79c ▷

```
freshVar : string -> term
```

```
local
  val n = ref 1
in
  fun freshVar s = VAR ("_" ^ s ^ intString (!n) before n := !n + 1)
end
```

Function `freshen` replaces free variables with fresh variables. Value `renaming` represents a renaming θ_α, as in Section D.3.4.

S79c. ⟨*renaming μProlog variables* S79b⟩+≡ (S593a) ◁ S79b

```
freshen : clause -> clause
```

```
fun freshen c =
  let val renamings = map (fn x => x |--> freshVar x) (clauseFreevars c)
      val renaming  = foldl compose idsubst renamings
  in  clausesubst renaming c
  end
```

Unification by solving equality constraints

To unify a goal with the head of a clause, we solve an equality constraint.

S79d. ⟨*substitution and unification* S78b⟩+≡ (S593a) ◁ S78b

```
unify : goal * goal -> subst
```

```
exception Unsatisfiable
⟨constraint solving (left as exercise)⟩
fun unify ((f, ts), (f', ts')) =
  solve (APPLY (f, ts) ~ APPLY (f', ts'))
```

As in Chapter 7, you implement the solver. Prolog uses the same kind of equality constraints as ML type inference, and it uses the same algorithm for the solver. If a constraint cannot be solved, `solve` must raise the `Unsatisfiable` exception.

S79e. ⟨*constraint solving* **⟦prototype⟧** S79e⟩≡

```
solve : con -> subst
```

```
fun solve c = raise LeftAsExercise "solve"
```

D.5.3 Backtracking search

I implement Prolog search using Byrd boxes (Section D.3.4, page S65), which are implemented in continuation-passing style. Given a goal g and continuations $\kappa_{\texttt{succ}}$ and $\kappa_{\texttt{fail}}$, `solveOne` g $\kappa_{\texttt{succ}}$ $\kappa_{\texttt{fail}}$ builds and runs a Byrd box for g. As expected for continuation-passing style, the result of the call to `solveOne` is the result of the entire computation.

Unless the predicate is built in, `solveOne` uses internal function `search` to manage the state of the Byrd box. Think of the argument to `search` as the list of clauses to be considered; the ⇒ arrow in Section D.3.4 points to the head of this list.[9]

To solve a single goal g using clause G `:-` $H_1, \ldots, H_m$, I rename variables, unify the renamed G with g to get θ, then solve $\theta(H_1), \ldots, \theta(H_m)$. Eventually, the entire composed substitution gets passed to $\kappa_{\texttt{succ}}$. In the code, $G =$ `conclusion` and $H_1, \ldots, H_m =$ `premises` (both after renaming), and $g =$ `goal`.

To solve multiple goals $g_1, \ldots, g_n$, I call `solveMany` $[g_1, \ldots, g_n]$ θ_{id} $\kappa_{\texttt{succ}}$ $\kappa_{\texttt{fail}}$, where θ_{id} is the identity substitution. Function `solveMany` manages interactions between Byrd boxes, composing substitutions as it goes. If substitution θ' solves goal g_1, we apply θ' to the remaining goals $g_2, \ldots, g_n$ before a recursive call to `solveMany`. If that recursive call fails, we transfer control to the `resume` continuation that came from solving g_1, which gives us a chance to produce a *different* substitution that might solve the whole lot.

The query search is coded as follows:

S80. ⟨*search* **⟦prototype⟧** S80⟩≡

```
query : database -> goal list  -> (subst -> (unit->'a) -> 'a) -> (unit->'a) -> 'a
solveOne  : goal          -> (subst -> (unit->'a) -> 'a) -> (unit->'a) -> 'a
solveMany : goal list -> subst -> (subst -> (unit->'a) -> 'a) -> (unit->'a) -> 'a
search   : clause list -> 'a
```

```
fun 'a query database =
  let val primitives = foldl (fn ((n, p), rho) => bind (n, p, rho))
                      emptyEnv  (⟨µProlog's primitive predicates :: S593c⟩ [])
      fun solveOne (goal as (predicate, args)) succ fail =
            find (predicate, primitives) args succ fail
            handle NotFound _ =>
              let fun search [] = fail ()
                    | search (clause :: clauses) =
                        let fun resume () = search clauses
                            val G :- Hs = freshen clause
                            val theta = unify (goal, G)
                        in  solveMany (map (goalsubst theta) Hs) theta succ resume
                        end
                        handle Unsatisfiable => search clauses
              in  search (potentialMatches (goal, database))
              end
      and solveMany []            theta succ fail = succ theta fail
        | solveMany (goal::goals) theta succ fail =
            solveOne goal
            (fn theta' => fn resume => solveMany (map (goalsubst theta') goals)
                                                 (compose (theta', theta))
                                                 succ
                                                 resume)
            fail
  in  fn gs => solveMany gs idsubst
  end
```

[9] Clauses preceding the ⇒ arrow are irrelevant to any future computation, and `search` discards them.

The environment `primitives` holds the primitive predicates. These predicates are implemented by polymorphic ML functions, and as a result, ML's "value restriction" prevents me from defining `primitives` at top level. To work around the restriction, function `query` rebuilds `primitives` once per query. Luckily the cost is small compared with the cost of the search.

D.5.4 Processing clauses and queries

μProlog's *basis* is the database of queries. μProlog uses the same generic read-eval-print loop as the other interpreters, which calls `processDef` on every definition. In μProlog, a "definition" is either a clause or a query.

S81a. ⟨*definitions of* `basis` *and* `processDef` *for μProlog* S81a⟩≡ (S593b)

```
type basis
processDef : cq * database * interactivity -> database
```

```
type basis = database
fun processDef (cq, database, interactivity) =
  let fun process (ADD_CLAUSE c) = addClause (c, database)
        | process (QUERY gs) = (⟨query goals gs against database S81b⟩; database)
      fun caught msg = (eprintln (stripAtLoc msg); database)
  in  withHandlers process cq caught
  end
```

To issue a query, I provide success and failure continuations to the `query` function defined above. The success continuation uses `showAndContinue` to decide between two possible next steps: resume the search and look for another solution, or just say "yes" and stop.

S81b. ⟨*query goals* `gs` *against* `database` S81b⟩≡ (S81a)

```
query database gs
  (fn theta => fn resume =>
     if showAndContinue interactivity theta gs then resume ()
      else print "yes\n")
  (fn () => print "no\n")
```

To show a solution, function `showAndContinue` applies the solution's substitution to the free variables of the query. Then, if the interpreter is prompting, `showAndContinue` waits for a line of input, and if the line begins with a semicolon, `showAndContinue` returns `true`, telling the interpreter to continue with the next solution. Otherwise it returns `false`. And if the interpreter isn't prompting, it's running batch mode, which produces at most one solution, so `showAndContinue` always returns `false`.

S81c. ⟨*interaction* S81c⟩≡ (S593b)

```
showAndContinue : interactivity -> subst -> goal list -> bool
```

```
fun showAndContinue interactivity theta gs =
  let fun varResult x = x ^ " = " ^ termString (varsubst theta x)
      val vars = foldr union' emptyset (map goalFreevars gs)
      val results = String.concatWith "\n" (map varResult vars)
  in  if null vars then
        false (* no more solutions possible; don't continue *)
      else
        ( print results
        ; if prompts interactivity then
            case Option.map explode (TextIO.inputLine TextIO.stdIn)
              of SOME (#";" :: _) => (print "\n"; true)
               | _ => false
          else
            (print "\n"; false)
        )
  end
```

ADD_CLAUSE	S55a
addClause	S78a
bind	305d
type clause	S54e
compose	S597c
type cq	S55a
emptyEnv	305a
emptyset	S217b
eprintln	S215b
find	305b
freshen	S79c
type goal	S54d
goalFreevars	S79a
goalsubst	S596e
idsubst	S596b
NotFound	305b
potentialMatches	S78a
prompts	S236c
QUERY	S55a
stripAtLoc	S235a
termString	S597d
unify	S79d
union'	S79a
Unsatisfiable	S79d
varsubst	S596c
withHandlers	S239a

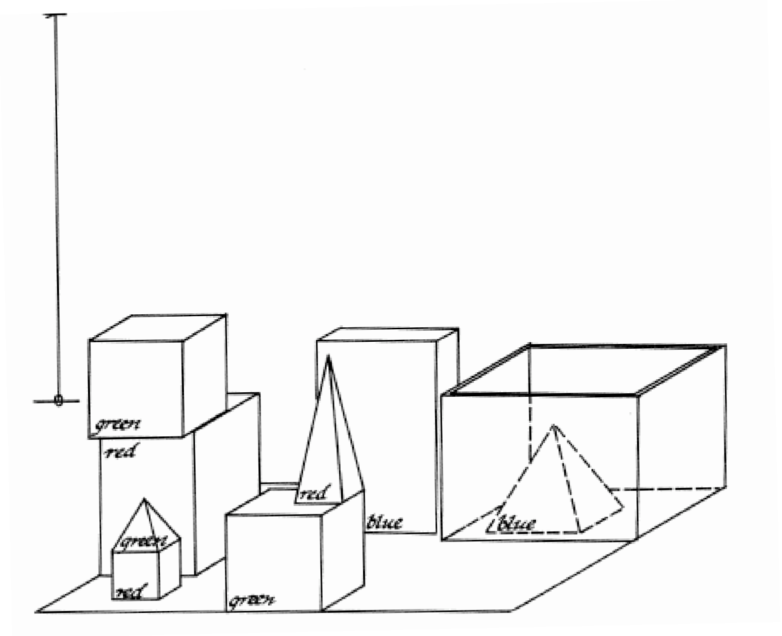


Figure D.4: The original blocks world as depicted by Winograd (1972)

D.6 Larger example: The blocks world

If you want to investigate language and reasoning, give your computer something simple to reason about. An idea that predates Prolog is to imagine discourse with a computer whose entire world consists of a table full of blocks (Figure D.4). The computer can see the blocks, and the computer controls a robot arm that can pick up and move one block at a time. This simple world was developed for one of the first language-understanding programs, SHRDLU. The blocks were designed "to give the system a world to talk about in which one can say many different kinds of things" (Winograd 1972, page 33).[10] In this example, we create Prolog axioms and inference rules for reasoning about blocks.

Even Winograd's blocks world is too complicated for a simple example, so let's consider a table containing only three cubical blocks labeled `a`, `b`, and `c`. And let's abstract away most of the details of the state—we don't care exactly where any block is located; all we want to know is what blocks are on top of what other blocks. Finally, let's not use natural language. Instead, let's use logic programming to tackle just one of the many problems solved by SHRDLU: developing a plan to get the blocks world from one state to another by moving one block at a time. For example, we might like to know how to get the blocks world from an initial state where each block is on the table to a desired state like that shown in Figure D.5 on page S85. We can tackle this problem using depth-first search; my design follows those of Kamin (1990, p. 362) and Sterling and Shapiro (1986, p. 222).

A key question is how to represent the state of the world. A state is determined by the answer to the question "what object is each block on top of?" We could, for

[10] Winograd's objective was the understanding of natural language, and while he was well informed of work in automated theorem proving using axioms and inference rules, he found it not practical enough to support language understanding or even reasoning about the blocks world. He observes that "logic is a declarative rather than imperative language, and to get an imperative effect requires a good deal of careful thought and clever trickery" (page 232). You are learning it.

example, represent a state as a three-tuple of objects. The initial state would be (table, table, table), and the desired state would be (b, table, a). But this state is hard to read. So instead of representing a state as a three-tuple, I use a list of relations:

State	*Representation*
Initial	`[on(a, table), on(b, table), on(c, table)]`
Desired	`[on(a, b), on(b, table), on(c, a)]`

We may as well allow relations to appear in any order, so two lists represent the same state if they contain the same relations.

The problem we're trying to solve is "given an arbitrary initial state, by what sequence of moves can we get to a desired state?" A "move" is the atomic action that the robot arm performs: it picks up a block from one place and sets it down in another. A move is represented by the term move(b, d), where b is a block and d is a destination.

To specify the effect of a move, we define our first predicate, which resembles a classic "Hoare triple": predicate triple(Pre, Move, Post) relates Move to states Pre and Post, which immediately precede and follow Move. Moving the first block in the state changes the thing the block is sitting on:

S83a. ⟨*transcript* S45a⟩+≡ ◁S77c S83b▷

```
?- [rule].
-> triple([on(Block, Thing) | S], move(Block, Dest), [on(Block, Dest) | S]).
```

Informally, if we move Block to Dest, the state changes so that instead of whatever Thing the Block was on before, it is now on Dest. But this rule works only if Block's location is the first relation in the state. What if the block occurs later? We need a rule that handles Block in other positions. Recursion seems promising, but we want to recur only if Block is *not* first. To guard the recursion, I use the same different predicate I use in the map-coloring problem.

S83b. ⟨*transcript* S45a⟩+≡ ◁S83a S83c▷

```
-> triple([on(B1, T1) | Pre], move(Block, Dest), [on(B1, T1) | Post]) :-
     different(Block, B1), triple(Pre, move(Block, Dest), Post).
```

Differences between blocks are made manifest in these axioms:

S83c. ⟨*transcript* S45a⟩+≡ ◁S83b S84a▷

```
-> different(a, b).  different(b, a).
-> different(a, c).  different(c, a).
-> different(b, c).  different(c, b).
```

Predicate triple tells how a move relates two states. It's a good predicate, but there's too much it doesn't know:

- You can't move a block to be on top of itself (a law of geometry).
- On the top of a cubical block, there is room for at most one other cubical block of the same size (geometry and physics).
- The robot arm can move a block, but it can't move the table.
- The robot arm can pick up a block only if nothing is on top of the block.

These facts are embodied in a new predicate legal_move.

Predicate legal_move can be proven with either of two inference rules. One rule moves a block onto the table, which can hold any number of blocks. The other rule moves a block onto another block, which can hold the first block only if no other block is on top of it. To say "in state S, nothing is on top of block B," I use the auxiliary predicate `holds_nothing(`B`, `S`)`.

S84a. ⟨*transcript* S45a⟩+≡ ◁S83c S84b▷

```
  -> block(a). block(b). block(c).  /* these things are blocks */
  -> legal_move(move(Block, table), S) :- block(Block), holds_nothing(Block, S).
  -> legal_move(move(B1, B2), S) :-
        block(B1), different(B1, B2), holds_nothing(B1, S), holds_nothing(B2, S).
```

A block holds nothing if nothing in the state is on it.

S84b. ⟨*transcript* S45a⟩+≡ ◁S84a S84c▷

```
  -> holds_nothing(Block1, [on(Block2, Thing) | S]) :-
        different(Block1, Thing), holds_nothing(Block1, S).
  -> holds_nothing(Block1, []).
```

This definition works only if the table is different from any block.

S84c. ⟨*transcript* S45a⟩+≡ ◁S84b S84d▷

```
  -> different(Block, table) :- block(Block).
  -> different(table, Block) :- block(Block).
```

A move might be legal and still not good. For example, a move might move a block to where it already is. Such a move is particularly bad because we are searching for a sequence of moves, and we can make arbitrarily many such moves without making progress. To rule out these useless moves, here is a predicate that is provable only if a move changes the state.

S84d. ⟨*transcript* S45a⟩+≡ ◁S84c S84e▷

```
  -> changes_state(move(Block, Dest), [on(Block, Thing) | S]) :- different(Dest, Thing).
  -> changes_state(move(Block, Dest), [on(B1, T1) | S]) :-
        different(Block, B1), changes_state(move(Block, Dest), S).
```

A move is good if it is legal and it changes state.

S84e. ⟨*transcript* S45a⟩+≡ ◁S84d S85b▷

```
  -> good_move(M, S) :- legal_move(M, S), changes_state(M, S).
```

We are now ready to search for a *sequence* of good moves that transforms one state into another. We might imagine we could compute such a list this way:

S84f. ⟨*nonterminating version of* `transforms` S84f⟩≡ S84g▷

```
  -> transforms(State, [], State).
  -> transforms(Initial, [Move|Moves], Final) :-
       good_move(Move, Initial),
       triple(Initial, Move, Intermediate),
       transforms(Intermediate, Moves, Final).
```

Regrettably, this idea won't work. For example, the following query asks for the transformation pictured in Figure D.5:

S84g. ⟨*nonterminating version of* `transforms` S84f⟩+≡ ◁S84f

```
  -> initial([on(a, b), on(b, table), on(c, a)]).
  -> desired([on(a, b), on(b, c), on(c, table)]).
  -> [query].
  ?- initial(S1), desired(S2), transforms(S1, Moves, S2).
```

The query does not terminate. To see why, let's add a `print` subgoal to the second clause of `transforms`:

S84h. ⟨*nonterminating version of* `transforms`*, with debugging code* S84h⟩≡

```
  -> transforms(Initial, [Move|Moves], Final) :-
       good_move(Move, Initial),
       triple(Initial, Move, Intermediate),
       print(moved(Move, Intermediate)),
       transforms(Intermediate, Moves, Final).
```

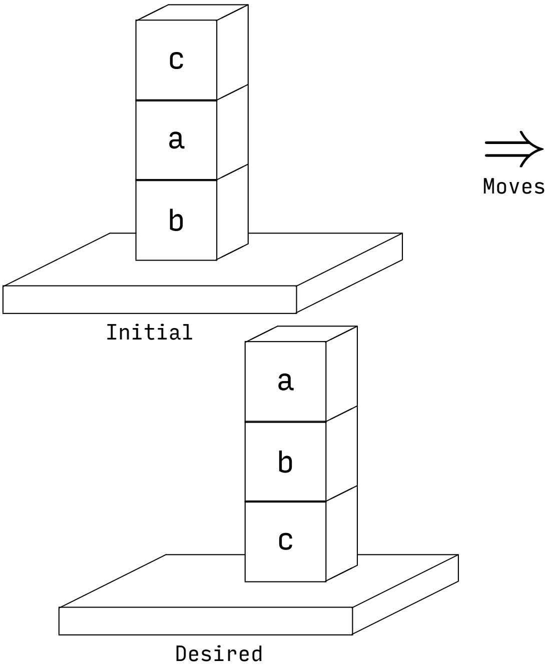


Figure D.5: Example problem in the simplified blocks world

(We can't just add this `print` subgoal, because we can't actually change an existing clause. All we can do is add new clauses to the database.[11] To work an example like this, we blow up our interactive session and start over with new definitions.)

The new `print` subgoal shows what is going on:

S85a. ⟨*output from nonterminating version of* `transforms`*, with debugging code* S85a⟩≡

```
moved(move(c, table), [on(a, b), on(b, table), on(c, table)])
moved(move(a, table), [on(a, table), on(b, table), on(c, table)])
moved(move(a, b), [on(a, b), on(b, table), on(c, table)])
moved(move(a, table), [on(a, table), on(b, table), on(c, table)])
moved(move(a, b), [on(a, b), on(b, table), on(c, table)])
 ...
```

The robot cheerfully puts block a on the table, then on block `b`, then back on the table, and so on forever. To stop it from visiting the same states repeatedly, we'll use depth-first search. The set of states already visited can live in an auxiliary variable, which becomes a *fourth* argument to the `transforms` predicate. The 4-argument version acts like an auxiliary function, and it can't possibly be confused with the three-argument `transforms`, because no substitution can make them equal. Predicate `transforms(Initial, Moves, Final, Visited)` holds if `Moves` leads from `Initial` to `Final` *without* passing through any state in `Visited`.

S85b. ⟨*transcript* S45a⟩+≡ ◁ S84e S86a ▷

```
-> transforms(State, [], State, Visited).
-> transforms(Initial, [Move|Moves], Final, Visited) :-
     good_move(Move, Initial),
     triple(Initial, Move, Intermediate),
     not_member(Intermediate, Visited),
     transforms(Intermediate, Moves, Final, [Intermediate|Visited]).
-> transforms(Initial, Moves, Final) :- transforms(Initial, Moves, Final, []).
```

[11] In full Prolog, you can remove a clause using the fancy predicate `retract`, but let's not go there—it's way too far outside the logical interpretation.

Predicate `not_member` does just what its name says.

S86a. ⟨*transcript* S45a⟩+≡ ◁S85b S86b▷

```
-> not_member(X, []).
-> not_member(X, [Y|YS]) :- different(X, Y), not_member(X, YS).
```

To make this code work, we extend `different` to states.

S86b. ⟨*transcript* S45a⟩+≡ ◁S86a S86c▷

```
-> different([on(A, X)|State1], [on(A, Y)|State2]) :- different(X, Y).
-> different([on(A, X)|State1], [on(A, X)|State2]) :- different(State1, State2).
```

With these new clauses, we get:

S86c. ⟨*transcript* S45a⟩+≡ ◁S86b S86d▷

```
-> initial([on(a, b), on(b, table), on(c, a)]).
-> desired([on(a, b), on(b, c), on(c, table)]).
-> [query].
?- initial(S1), desired(S2), transforms(S1, Moves, S2).
S1 = [on(a, b), on(b, table), on(c, a)]
S2 = [on(a, b), on(b, c), on(c, table)]
Moves = [move(c, table), move(a, table), move(b, a), move(b, c), move(a, b)]
yes
```

The plan works, but it's not great. Moving block `b` twice in a row is not smart. Eliminating double moves helps (Exercise 21, page S103), but we can do even better.

To do better, let's reconsider what step to take from an `Initial` state. In this step, predicate `transforms` does not take the `Final` state into account. To direct the search, let's define a new predicate `better_move(Move, Initial, Final)`, which prefers moves that move us closer to the `Final` state. Predicate `transforms2` is like `transforms`, except it uses `better_move` instead of `good_move`.

S86d. ⟨*transcript* S45a⟩+≡ ◁S86c S86e▷

```
?- [rule].
-> transforms2(State, [], State, Visited).
-> transforms2(Initial, [Move|Moves], Final, Visited) :-
     better_move(Move, Initial, Final),
     triple(Initial, Move, Intermediate),
     not_member(Intermediate, Visited),
     transforms2(Intermediate, Moves, Final, [Intermediate|Visited]).
-> transforms2(Initial, Moves, Final) :- transforms2(Initial, Moves, Final, []).
```

Predicate `better_move` in turn uses `suggest`, which looks at `Final` and suggests moving a block directly to the location where it is in the `Final` state.

S86e. ⟨*transcript* S45a⟩+≡ ◁S86d S86f▷

```
-> better_move(Move, Initial, Final) :- suggest(Move, Final),
                                        good_move(Move, Initial).
-> better_move(Move, Initial, Final) :- good_move(Move, Initial).
-> suggest(move(Block, Dest), State) :- member(on(Block, Dest), State).
```

The suggestion eliminates the double move:

S86f. ⟨*transcript* S45a⟩+≡ ◁S86e S87▷

```
-> [query].
?- initial(S1), desired(S2), transforms2(S1, Moves, S2).
S1 = [on(a, b), on(b, table), on(c, a)]
S2 = [on(a, b), on(b, c), on(c, table)]
Moves = [move(c, table), move(a, table), move(b, c), move(a, b)]
yes
```

In fact, this plan is optimal: getting from `S1` to `S2` requires at least four moves.

D.7 Larger example: Haskell type classes

Logic programming is a key ingredient in the type system of the popular functional language Haskell. Logic programming is part of Haskell's system of *type classes*, which determines the meanings of names like `==` (equality), `<` (comparison), `+` (arithmetic), and `show` (printing). Each of these operations has a type that uses *bounded polymorphism* (Chapter 9); the operation can be used at any type that meets a constraint:

Operation	*Type*
`==`	`(forall ['a where (Eq 'a)]   ('a 'a -> bool))`
`<`	`(forall ['a where (Ord 'a)]  ('a 'a -> bool))`
`+`	`(forall ['a where (Num 'a)]  ('a 'a -> 'a))`
`show`	`(forall ['a where (Show 'a)] ('a 'a -> string))`

(The types are written not as they are in Haskell but as they might be written in an extension of Typed μScheme or Molecule.)

Logic programming enters the picture in two ways:

- Haskell uses a logic program to to prove that constraints like `Eq (list int)` are satisfied.
- Haskell also uses a logic program *to generate code* for the instance of `==` at type `(list int)`. The generated implementation of `==` provides constructive evidence that `Eq (list int)` is satisfied; it is sometimes called a *witness*. This ability to generate code from a type is one of Haskell's mutant superpowers (Claessen and Hughes 2000).

This section develops the example by providing inference rules for a single predicate,

$$\texttt{implemented\_by}(O, T, F),$$

which holds when function F implements the instance of polymorphic, overloaded operation O at type T. Making a query at a given O and T produces the generated function F.

To represent the names of operations, I use Prolog functors. To represent Haskell's expressions and types, I use Prolog terms. How terms can represent Haskell expressions and types is a question that cannot be answered in Prolog itself, but I can specify informally which terms represent types. One of the simplest and best specifications is a grammar.[12]

htype ::= `int` | `bool` | `pairtype(`*htype*`,`*htype*`)` | `listtype(`*htype*`)`
 | `arrowtype([`{*htype*`,`}`],`*htype*`)`
hexp ::= x | `lambda([`{`arg(`x`,`*htype*`),`}`],`*hexp*`)` | `apply(`*hexp*`,[`{*hexp*`,`}`])`
 | `if(`*hexp*`,`*hexp*`,`*hexp*`)` | `letrec(`x`,`*hexp*`,`*hexp*`)`

In addition, I assume the existence of primitive functions for comparison on base types (`inteq`, `intlt`), for introducing and eliminating pairs (`pair`, `fst`, `snd`), and for operating on lists (`isnull`, `cons`, `car`, `cdr`). Finally, to spell Haskell's operators in Prolog, instead of `==`, `<`, and `+` I write `eq`, `lt`, and `plus`.

I begin my proof system with a claim that integers can be compared for equality, and the function to be used is `inteq`.

S87. ⟨*transcript* S45a⟩+≡ ◁ S86f S88a ▷

```
?- [rule].
-> implemented_by(eq, int, inteq).
```

[12] Warning: at the end of each list, the grammar shows a specious comma.

And integers can be compared for order.

S88a. ⟨*transcript* S45a⟩+≡ ◁ S87 S88b ▷

```
-> implemented_by(lt, int, intlt).
```

To compare Booleans for equality, I use the function

```
(lambda ([p : bool] [q : bool]) (if p q (not q)))
```

In Prolog, the function is encoded by a term:

S88b. ⟨*transcript* S45a⟩+≡ ◁ S88a S88c ▷

```
-> implemented_by(eq,
                  bool,
                  lambda([arg(p,bool),arg(q,bool)],if(p,p,apply(not,[q])))).
```

I order Booleans by putting falsehood before truth, so my `lt` function is

```
(lambda ([p : bool] [q : bool]) (if p #f q))
```

S88c. ⟨*transcript* S45a⟩+≡ ◁ S88b S88d ▷

```
-> implemented_by(lt, bool, lambda([arg(p,bool),arg(q,bool)],if(p,false,q))).
```

Now let's generate some code. I start by generating code to compare pairs of types τ_1 and τ_2. Two pairs are equal if both their elements are equal, so I need two equality functions $=_1$ and $=_2$. Given those functions, I compare pairs `p1` and `p2` using this function:

```
(lambda ([p1 : $\tau_1$] [p2 : $\tau_2$])
    (if ($=_1$ (fst p1) (fst p2))
        ($=_2$ (snd p1) (snd p2))
        #f))
```

Here it is in Prolog:

S88d. ⟨*transcript* S45a⟩+≡ ◁ S88c S88e ▷

```
-> implemented_by(eq, pairtype(T1, T2),
                  lambda([arg(p1, pairtype(T1,T2)),
                          arg(p2, pairtype(T1,T2))],
                         if(apply(EQ1,[apply(fst,[p1]),apply(fst,[p2])]),
                            apply(EQ2,[apply(snd,[p1]),apply(snd,[p2])]),
                            false))) :-
      implemented_by(eq, T1, EQ1),
      implemented_by(eq, T2, EQ2).
```

At this point I can ask, for example, for a function used to compare pairs of type `(pair int bool)`:

S88e. ⟨*transcript* S45a⟩+≡ ◁ S88d S89a ▷

```
-> [query].
?- implemented_by(eq, pairtype(int, bool), EQIB).
EQIB = lambda([arg(p1, pairtype(int, bool)), ...
yes
```

The full definition of EQIB is a snarl that only a compiler writer could love, but it can be prettyprinted into something a programmer would recognize:

```
(lambda ([p1 : (pair int bool)] [p2 : (pair int bool)])
    (if (inteq (fst p1) (fst p2))
        ((lambda ([p : bool] [q : bool]) (if p p (not q)))
            (snd p1)
            (snd p2))
        #f))
```

This code could be simplified—the inner `lambda` is applied to known arguments—but any compiler for any functional language includes a simplifier that is more than capable of dealing with such code.

As another example, here is < on pairs. Haskell allows < only when it also has equality, so I assume the same.

S89a. ⟨*transcript* S45a⟩+≡ ◁S88e S89b▷

```
?- [rule].
-> implemented_by(lt, pairtype(T1, T2),
                  lambda([arg(p1, pairtype(T1,T2)),
                          arg(p2, pairtype(T1,T2))],
                         if(apply(EQ1,[apply(fst,[p1]),apply(fst,[p2])]),
                            apply(LT2,[apply(snd,[p1]),apply(snd,[p2])]),
                            apply(LT1,[apply(fst,[p1]),apply(fst,[p2])])))) :-
      implemented_by(eq, T1, EQ1),
      implemented_by(lt, T1, LT1),
      implemented_by(lt, T2, LT2).
```

We can now ask for < on, for example, a pair of integers:

S89b. ⟨*transcript* S45a⟩+≡ ◁S89a S89c▷

```
-> [query].
?- implemented_by(lt, pairtype(int, int), LTII).
LTII = lambda([arg(p1, pairtype(int, int)), ...
yes
```

The code bound to `LTII` prettyprints as follows:

```
(lambda ([p1 : (pair int int)] [p2 : (pair int int)])
    (if (inteq (fst p1) (fst p2))
        (intlt (snd p1) (snd p2))
        (intlt (fst p1) (fst p2))))
```

Let's wrap up by generating a recursive function. If we have function $=_\tau$ for comparing list elements, we can compare lists using this function:

```
(letrec ([eqlists (lambda ([xs : (list τ)] [ys : (list τ)])
                      (if (null? xs)
                          (null? ys)
                          (if (null? ys)
                              #f
                              (if (=_τ (car xs) (car ys))
                                  (eqlists (cdr xs) (cdr ys))
                                  #f))))])
  eqlists)
```

That rule is coded in μProlog as follows:

S89c. ⟨*transcript* S45a⟩+≡ ◁S89b S90a▷

```
?- [rule].
-> implemented_by(eq, listtype(T),
     letrec(eqlists,
        lambda([arg(xs, listtype(T)), arg(ys, listtype(T))],
               if(apply(isnull,[xs]),
                  apply(isnull,[ys]),
                  if(apply(isnull,[ys]),
                     false,
                     if(apply(EQT,    [apply(car,[xs]),apply(car,[ys])]),
                        apply(eqlists,[apply(cdr,[xs]),apply(cdr,[ys])]),
                        false)))),
        eqlists)) :-
    implemented_by(eq, T, EQT).
```

All the examples above imitate what Haskell does with its type-class system. Each rule for predicate `implemented_by` corresponds to a Haskell *instance declaration*. But with Prolog, we can do more. For example, we can define ML's notion of a type that "admits equality." A type admits equality if there is an implementation of `eq`.

S90a. ⟨*transcript* S45a⟩+≡ ◁S89c S90b▷

```
-> admits_equality(T) :- implemented_by(eq, T, F).
```

Here, as in ML, types emit equality as long as no function types are involved.

S90b. ⟨*transcript* S45a⟩+≡ ◁S90a S92a▷

```
-> [query].
?- admits_equality(int).
yes
?- admits_equality(listtype(pairtype(int, listtype(int)))).
yes
?- admits_equality(arrowtype([int, int], bool)).
no
```

D.8 Prolog as it really is

D.8.1 Syntax

μProlog's syntax is close to the syntax of the ISO standard; both are based on Edinburgh Prolog (Clocksin and Mellish 2013). Full Prolog allows additional control structures in clauses and queries, of which the most notable are disjunction, written with a semicolon, and conditional, written `(`g_1 `->` g_2`;` g_3`)`.

Real Prolog uses different naming conventions than μProlog. In μProlog, I use past participles such as `reversed`, `appended`, `sorted`, and so on. I do so in order to emphasize the distinction between programming with predicates and programming with functions. In full Prolog, it is more idiomatic to name one's predicates using imperative verb forms such as `reverse`, `append`, and `sort`.

D.8.2 Logical interpretation as a single first-order formula

Section D.3.4 describes logical interpretation of Prolog in terms of proofs and derivations. Left unspecified is what algorithm to use to find a proof. But Prolog was invented in part to take advantage of one particular algorithm: the *resolution* technique invented by Robinson (1965). The details are beyond the scope of this book, but in this section I sketch the ideas.

The first idea is that a Prolog query can be viewed purely as a question about a formula in first-order logic, with no need to construct a derivation. The key to this view is that every Prolog clause corresponds to a first-order formula:

$$\begin{aligned} G : - H_1, \ldots, H_n &\equiv H_1 \wedge \cdots \wedge H_n \implies G \\ &\equiv \neg(H_1 \wedge \cdots \wedge H_n) \vee G \\ &\equiv \neg H_1 \vee \cdots \vee \neg H_n \vee G. \end{aligned}$$

Let us write this last formula as C, and let us imagine that C is wrapped in a universal quantifier $\forall X_1, \ldots, X_k$, where $X_1, \ldots, X_k$ are the free variables of the clause.

The entire database can be viewed as the conjunction of all the clauses: $C_1 \wedge \ldots \wedge C_m$. By a suitable renaming of variables, we can pull all the universal quantifiers out to the front. Writing $\vec{X}$ for the list of all the logical variables mentioned in the database, we can say

$$D = \forall \vec{X} : C_1 \wedge \ldots \wedge C_m.$$

In the jargon of mathematical logic, the database is a *closed, first-order formula*.

When we write a query $g_1, \ldots, g_j$, we are asking if there *exists* an assignment to variables of the g's such that the database implies all the g's. Writing $\vec{Y}$ for the list of all the logical variables that appear in $g_1, \ldots, g_j$, we are asking about the formula

$$(\forall \vec{X} : C_1 \wedge \ldots \wedge C_m) \implies \exists \vec{Y} : g_1 \wedge \cdots \wedge g_j,$$

which is another closed, first-order formula. What we want to know is if this formula is *valid*—that is, given any sensible interpretation of predicates as relations, functors as functions, and atoms as objects, is the formula true? And in classical logic, a first-order formula is valid if and only if its complement leads to a contradiction—that is, if the complement can be *refuted*.

The complement of our formula is

$$\begin{aligned} F &= \neg((\forall \vec{X} : C_1 \wedge \ldots \wedge C_m) \implies \exists \vec{Y} : g_1 \wedge \cdots \wedge g_j) \\ &\equiv \forall \vec{X} : C_1 \wedge \ldots \wedge C_m \wedge \forall \vec{Y} : \neg(g_1 \wedge \cdots \wedge g_j) \\ &\equiv \forall \vec{X} : \forall \vec{Y} : C_1 \wedge \ldots \wedge C_m \wedge (\neg g_1 \vee \cdots \vee \neg g_j). \end{aligned}$$

If F can be refuted, there is a particular assignment to the $\vec{Y}$ that refute the inner formula. These $\vec{Y}$ satisfy the query.

This presentation should seem very abstract. To connect it to Prolog requires a genius like Robinson. Formula F is a conjunction of disjunctions, also known as *conjunctive normal form*. Robinson's *resolution* method discovers refutations of formulas in conjunctive normal form. Resolution matches $\neg H_i$'s and $\neg g_i$'s, which have logical complement $\neg$ in front of them, with G's, which don't have a logical complement. If you revisit the individual formulas that are conjoined together, you can verify that in any one conjunct, at most one predicate is *not* complemented. That property makes resolution very effective, because for any given $\neg g_i$ or $\neg H_i$, there is at most one candidate G in each conjunct. The details of resolution are beyond the scope of this book, but are explained well by Kamin (1990, Chapter 8).

To return to Prolog, the g_i's are goals in the query, the H_i's are subgoals, and each G is the head of some clause. The "matching" performed by resolution is actually unification. And the property that in each conjunct, at most one predicate is not complemented? That property is built into Prolog's design, on purpose. The property is so important that it has a name: this form of formula is called a *Horn clause*.

This second logical interpretation of Prolog says that making a query is equivalent to building a *single* logical formula that says "for all X's in the database, the assertions in the database imply that there exist a set of Y's such that the query is satisfied." This interpretation is elegant, and it is supported by Robinson's efficient resolution algorithm. But it is a little more difficult to connect to what actually goes on in a Prolog interpreter, and for the beginning Prolog programmer it is of more historical and academic interest than practical interest.

D.8.3 Semantics

Full Prolog is a nice, simple language, and its semantics is largely the same as the semantics of μProlog, but with some powerful extensions. The most important extensions are the "cut" and `not`. Full Prolog also has a large initial basis which includes not only input/output and arithmetic but also many predicates that reflect on the state of the Prolog machine and the computation itself. We look at two of the relatively easy and interesting reflective predicates, `assert` and `retract`.

The occurs check

The most salient difference between full Prolog and μProlog is that implementations of full Prolog typically omit the *occurs check* (page S60), at least by default. The occurs check takes time linear in the size of a term, so omitting it can save a lot, reducing some algorithms from quadratic time to linear time. But when the occurs check is omitted, the programmer is obligated to avoid unifying a variable with a term which contains that variable—or to use run-time flags or predicates that reinstate the occurs check. If you take Prolog seriously, it is an obligation to be aware of.

Extra-logical features: The cut and the `not` *predicate*

The extension called the *cut* limits backtracking. A cut is written by using the exclamation mark (`!`) as a goal. A clause with a cut takes the form

$$G \texttt{ :- } H\texttt{, !, } H'\texttt{.}$$

When this clause is used, it is to try to satisfy goal g with which the head G unifies. In the usual way, the search tries to satisfy subgoal H, then the cut, then H'. An attempt to prove a cut always succeeds; that is, a cut is always satisfied. If subgoals H and H' are also satisfied, g is proven, and the cut plays no substantial role. If H cannot be satisfied, the search never arrives at the cut, and again it plays no role. But if H is satisfied, and then (because the cut is always satsified) H' cannot be satisfied, the search backtracks. And when it backtracks into the cut, it does *not* continue by trying to find a different substitution that proves H. Instead, backtracking into the cut causes the goal g to fail immediately. Goal g fails even if there are later clauses in the database that might apply to g.

The cut simplifies many computations that involve some sort of negation. An example is this definition of `not_equal`:

S92a. ⟨*transcript* S45a⟩+≡ ◁S90b S92b▷

```
?- [rule].
-> not_equal(X,Y) :- equal(X,Y), !, fail.
-> not_equal(X,Y).
```

where the definition of `equal` is the single clause:

S92b. ⟨*transcript* S45a⟩+≡ ◁S92a

```
-> equal(X,X).
```

Predicate `not_equal(X,Y)` makes sense only when `X` and `Y` are bound to ground terms. When `X` and `Y` are unequal, `not_equal(X,Y)` is satisfied. When `X` and `Y` are equal, `not_equal(X,Y)` is unsatisfiable.

As an example, query `not_equal(1, 2)` triggers these computational steps:

1. The query matches the first clause with `X = 1` and `Y = 2`. The first subgoal on the right-hand side is therefore `equal(1, 2)`. Because 1 is not identical to 2, that subgoal fails, and Prolog backtracks, looking for another clause that matches query `not_equal(1, 2)`.

2. The query matches the second clause with `X = 1` and `Y = 2`. There are no subgoals, to the original query is satisfied: Prolog proves `not_equal(1, 2)`.

Compare that computation with what ensures after query is `not_equal(2, 2)`:

1. The query matches the first clause with `X = 2` and `Y = 2`. The first subgoal is therefore `equal(2, 2)`. Because 2 is identical to 2, `equal(2, 2)` succeeds.

2. The next subgoal from the first clause is the cut, which always succeeds in the forward direction.

3. The next and final subgoal from the first clause is `fail`. Predicate `fail/0` is a conventional predicate that can't be proven; it always fails.

4. Now Prolog backtracks into the cut. This backtracking causes the original query, `not_equal(2, 2)`, to fail.

In both cases, Prolog proves what we expect.

The idiom of "cut-then-fail" can be used with many predicates. For example, the `not_member` predicate from the blocks world can be defined using

```
not_member(X,Y) :- member(X,Y), !, fail.
not_member(X,Y).
```

The idiom is so common that Prolog provides an implementation using the primitive predicate `not`. Using this predicate, we can write

```
not_member(X,Y) :- not(member(X,Y)).
```

The predicate `not` is a special *reflective* predicate. Its argument is not just a term; its argument is a fragment of a Prolog program—in this case, a goal. Query `not(`g`)` asks a question about computing with goal g: is it provable? If g is provable, query `not(`g`)` fails. If g is *not* provable, query `not(`g`)` succeeds. This behavior is called "negation as failure"; it is another example of how Prolog deals in provability, not in truth.

Prolog's `not` also upends the logical interpretation. Our normal idea of a query is "can we find a substitution for the logical variables such that the resulting proposition is provable?" For example, the query `not(member(X, [2, 4, 6]))` might stand for a logical formula like $\exists X : \neg(X \in \{2, 4, 6\})$, to which the answer is yes, there is an X not in $\{2, 4, 6\}$—in fact there are infinitely many. But when we issue that query to Prolog, the logical question that is actually being asked is if there exists an X that makes $X \in \{2, 4, 6\}$ provable, and the answer to *that* question is also yes, so the answer to the `not` query is no. The difference is the difference between two formulas:

What you might think you are asking $\exists X : \neg(X \in \{2, 4, 6\})$,
What you are actually asking $\neg(\exists X : X \in \{2, 4, 6\})$.

This contrast suggests a heuristic for working with `not`: to avoid confusion about where the existential quantifier goes, make sure there is no existential quantifier. In other words, ask `not(`g`)` only when g is a ground term.

In addition to its role in negation, the cut can also be used for efficiency: when an early goal is proven without substituting for any logical variables, but a later goal fails, there is no need to search for a second proof of the early goal. To see an example, imagine this generic query:

```
generate(X), member(X, zs), test(X)
```

with these assumptions:

1. Goal `generate(X)` succeeds only by substituting a ground term for `X`. But it is likely to succeed multiple times with multiple different `X`'s, just like the goal `better_move(X, Initial, Final)` in Section D.6.

2. Term zs is a ground term. Because both `X` and zs are ground terms, the subgoal `member(X, `zs`)` is executed only for success or failure—it never substitutes for a logical variable.

3. Sometimes `test(X)` succeeds and sometimes it fails.

Now imagine what happens if `member` is defined as on page S51. If `generate` and `member` succeed but `test` fails, backtracking will cause `member` to search the *entire* list zs. But this search is wasted effort: whether it succeeds or fails, it can't change `X`. This kind of wasted effort can be eliminated by using the cut, as in this revised definition of `member`:

```
member(X,[X|XS]) :- !.
member(X,[Y|YS]) :- member(X,YS).
```

Once `member` is defined in this way, any backtracking into `member` aborts immediately, and backtracking resumes with `generate(X)`. I think of this use of the cut as enforcing "succeed at most once."

The correctness of the succeed-at-most-once trick rests on a long chain of assumptions, and it throws the logical interpretation out the window. The cost of the performance improvement is a significant change in the semantics of `member`. For example, in the new semantics, if `X` is *not* instantiated to a ground term, the query `member(X, [1, 2, 3])` means exactly the same thing as the query `equal(X, 1)`. Not what you hoped for. But sometimes, to get a Prolog program to perform well, you really do want the cut.

Both the cut and the primitive `not` predicate are easy to add to μProlog (Exercises 44 and 45, page S112).

Changing the database: `assert` *and* `retract`

Another reflective feature of Prolog is provided by predicates `assert` and `retract`, which enable a program to add clauses to or remove clauses from the database. Each of these predicates takes a clause as its argument. These predicates are like `print`: an attempt to prove one always succeeds, and success has a side effect:

- Predicate `assert(`C`)` places C into the database, at a position that is not specified. Variants `asserta` and `assertz` put C in first and last positions, respectively.

- Predicate `retract(`C`)` finds and removes the first clause in the database that matches C.

These predicates can add or remove any any clause, but a common use is to simulate the effect of a global variable. For example, let's suppose that you want to instrument a blocks-world program to count the total number of moves generated, which I'll call N. This information can be represented by storing a single clause in the database of the form `moves_generated(`N`)`. The counter can be initialized by defining

```
moves_generated(0).
```

The number of moves can be incremented by predicate `bump_moves`, defined as follows:

```
bump_moves :- retract(moves_generated(N)),
              M is N+1,
              assert(moves_generated(M)).
```

To reset the counter, use predicate `reset_moves`:

```
reset_moves :- retract(moves_generated(X)), assert(moves_generated(0)).
```

A more interesting use of `assert` and `retract` is to convert data into code. Exercise 47 (b) on page S113 asks you to use `assert` and `retract` to convert map-coloring *data* into a map-coloring *rule*. This model enables a skilled Prolog programmer to avoid the layer of interpretation required by Exercise 7.

Full Prolog versus logic

Primitive predicates `assert` and `retract`, as well as `not` and the cut, cannot be explained in logic—they make sense only when viewed through the procedural interpretation of Prolog. In full Prolog, many other primitive predicates are the same way. This aspect of Prolog is viewed as a major weakness: the logical interpretation doesn't describe the full language, and the procedural interpretation, even with the help of Byrd boxes, is too hard to understand. An ideal language for logic programming would have programs that make sense in logic, and some other way to manage the database and the search for proofs. As Robinson (1983) put it, "we ought not to incorporate into the logical notation itself particular conventions about how to manage the details of the deductive search." For better or worse, Robinson's view has not carried the day; serious Prolog programmers know that they can't treat Prolog as simple first-order logic, and they expect to use non-logical features, including reflection and the cut.

D.9 Summary

In logic programming, we solve problems using predicates, propositions, formulas, and terms. Symbols for functions and values exist, but except for simple arithmetic, the functions and values are unspecified. Atoms and functors act like value constructors in ML: an atom is identical to itself, and identical functors applied to identical arguments produce identical results. A logic program takes a set of asserted formulas, both facts and rules, and asks what is provable—not necessarily what is true.

The best-known exemplar of logic programming is Prolog. It has proponents in a wide variety of fields, but is probably best known for use in artificial intelligence, natural-language processing, and expert systems. You can find Prolog in unexpected places, however; my two favorites are the first interpreter for Erlang and the operating-system bootstrap code used in Microsoft Windows NT.

D.9.1 Key words and phrases

Atom A Prolog object consisting of a single name, like `jacques` or `yellow`. Like a Scheme atom, its only property is that it is identical to itself.

Clause A valid reasoning principle stored in the Prolog database, consisting of a *conclusion* or *head* that is justified by means of zero or more *premises*. If there are no premises, the clause is called a fact; otherwise it is a rule. If a clause contains logical variables, they are implicitly universally quantified. That is, any term may be substituted for any variable, and the resulting rule is considered a valid reasoning principle.

Completeness An algorithm for implementing logic programs is called *complete* if, whenever a proof of a query exists, the algorithm eventually finds such a proof. As a system for proving that a formula implies a contradiction, the algorithm used by Prolog, *resolution,* is complete. Prolog's search algorithm is not complete.

A logic itself is called complete if every judgment that is true in all models is also provable.

Decidability A question is called *decidable* if there is an algorithm for answering it that is sound, complete, and *terminating* on all inputs. In propositional logic, the general query problem "is this formula provable?" is decidable. (One decision procedure is to enumerate the truth table of the formula; this procedure works because propositional logic is sound and complete with respect to the model of truth tables.) In general first-order logic, the general query problem "is this formula provable?" is *not* decidable.

Fact A proposition asserted as fact and entered into the Prolog database. If it contains logical variables, they are implicitly universally quantified.

Goal A proposition, or conjunction of propositions, that Prolog tries to prove using clauses. Prolog's proof process may substitute for logical variables in the goal.

Ground term A term that contains no logical variables.

Logic programming A style of programming in which a program is regarded as an assertion in a logic, and a computation asks whether a given query is *provable* from the assertions in the program.

Logical variable In first-order logic, a variable that may stand for a mathematical object drawn from some domain. In Prolog, a variable that may stand for a term—or for which a term may be substituted. Unlike a variable in an imperative language, whose value is set by assignment, or a variable in a functional language, whose value is bound by function application or `let` binding, a logical variable is associated with a value by means of a substitution, usually computed by unification.

Model A model of a language is a mapping from each symbol of the language to a mathematical object. Objects are made up of a *universe*, which is a nonempty set A. Function symbols, like Prolog functors, map to functions. Predicate symbols map to relations; a predicate symbol of arity n maps to a subset of the Cartesian product space A^n.

Object What a variable may stand for in logic; a thing from a (mathematical) domain.

Occurs check The part of unification that refuses to unify a variable `X` with a non-variable term t whenever `X` occurs in t. The occurs check guarantees that the substitution returned by unification does indeed solve the given equality constraint. If the occurs check is omitted, the underlying logic may be made unsound. However, the occurs check is perceived as expensive, and popular implementations of full Prolog omit it by default. Making sure the resulting program is sound is up to the programmer (who may instead choose to turn on the occurs check).

Predicate The means of forming propositions. A zero-place predicate is a proposition by itself; a multi-place predicate forms a proposition when applied to one or more terms. In Prolog, a predicate is identified by the combination of its symbol (an atom) and the number of arguments to which it is applied, as in `member/2` or `person/1`.

Predicate logic An extension of propositional logic that allows for logical variables to be quantified using the universal and existential quantifiers $\forall$ and $\exists$. In first-order logic, a variable may stand only for a mathematical object. In second-order logic, a variable may stand for a predicate or function. First-order predicate logic is not decidable, but when a proof of a formula exists, there are sound and complete algorithms for discovering it.

Property Convenient shorthand for a one-place predicate.

PROPOSITION The fundamental unit of propositional logic (that is, logic without quantifiers). In Prolog, a PREDICATE applied to zero or more arguments.

PROPOSITIONAL LOGIC A language of uninterpreted propositions and logical connectives. There are several popular sets of connectives, all equivalent. One minimal set is implication $\Longrightarrow$ and negation $\neg$. Another popular set is conjunction $\wedge$, disjunction $\vee$, and negation $\neg$—possibly augmented with implication. All these sets are equivalent to the singleton set containing only the NAND operator, where $x \text{ NAND } y = \neg(x \wedge y)$. Propositional logic is DECIDABLE.

QUERY A GOAL posed to the Prolog engine at top level. If it contains logical variables, they are implicitly existentially quantified—at least in the logical interpretation of Prolog.

RELATION Convenient shorthand for a PREDICATE of two or more places. Also, the species of mathematical object that a predicate stands for.

RULE An inference rule asserted as valied and entered into the Prolog database. Contains a *conclusion* (also called *head*) and one or more *premises*, all of which are propositions. If a rule contains logical variables, they are implicitly universally quantified.

SOUNDNESS An algorithm for implementing logic programs is called *sound* if, whenever the algorithm says a judgment is provable, the judgment is actually provable in the logic. The algorithm used by Prolog, *resolution*, is sound, but omitting the OCCURS CHECK can make it unsound. A logic itself is called sound if every provable judgment is true in all MODELS.

SUBGOAL A subsidiary GOAL spawned by Prolog's proof search. Also, one conjunct in a goal that is a conjunction.

SUBSTITUTION A finite mapping from LOGICAL VARIABLES to TERMS. Extends to structure-preserving mappings on terms and CLAUSES.

TERM Prolog's representation of a mathematical object: an atom, a number, or a functor applied to one or more terms.

UNIFICATION The algorithm used to discover a substitution θ that makes two terms identical—that is, the algorithm used to find a solution to an equality constraint $t_1 \sim t_2$.

D.9.2 Further Reading

While it is usually fun to go to the source, the original report on Prolog is written in French (Colmerauer et al. 1973). A good alternative is an early article by Kowalski (1974). Although the article opens with some startling claims about "human logic" versus "mathematical logic"—as if mathematicians weren't human—it proceeds to lay out the logic-programming agenda nicely, and it explains Horn clauses, which are the logical basis for the form of clauses that Prolog accepts.

Retrospective commentary about Prolog can be found in an address by Robinson (1983), who identifies many contributors, and who also pleads with his audience for a principled approach to the subject. Another retrospective, from Cohen (1988), describes applications in natural-language processing and in automated theorem proving, and it compares the development of Prolog with the development of Lisp. Kowalski (1988) presents a more personal retrospective, focusing on developments at Edinburgh in the 1970s. His presentation includes comparisons between logic programs and the PLANNER approach used by Winograd (1972) in his work on the original blocks world.

As suggested in Section D.1, logic programming encourages a different way of thinking about programming. Kowalski (1979, 2014) introduces logic, computer programming, and problem-solving at book length, for an audience of beginners; I recommend this book highly.

The standard introduction to Prolog is by Clocksin and Mellish (2013). There are other introductory texts by Hogger (1984) and Sterling and Shapiro (1986).

The Byrd box was originally proposed as a conceptual tool for understanding Prolog, not as an implementation technique (Byrd 1980). Proebsting (1997) shows how to use Byrd boxes to implement Icon, another language that has backtracking built in (Griswold and Griswold 1996).

Efficient implementation of Prolog rests on two technologies. The *resolution* principle (Robinson 1965) offers an algorithm for refuting formulas in conjunctive normal form; when formulas are limited to Horn clauses (Exercise 11, page S101), the asymptotic costs of resolution are made tractable. Warren (1983) proposes an abstract machine, including an instruction set, for executing Prolog programs; this machine has informed many efficient implementations. If you want to understand Warren's abstract machine, consult one of the tutorial presentations by Kogge (1990) or Aït-Kaci (1991).

To the best of my knowledge, the blocks world was created by Winograd (1972) for his doctoral work on language understanding. Winograd's dissertation reflects the 1970s belief, strongly held in North America, that approaches based only on logic would not be sufficient for understanding natural language. The blocks world appears in many books on artificial intelligence (Winograd 1972; Winston 1977; Nilsson 1980) and on logic programming (Kowalski 1979; Sterling and Shapiro 1986). My solution to the moves problem is derived from those of Kamin (1990) and Sterling and Shapiro (1986).

D.10 Exercises

Highlights

Here are some of the highlights of the exercises below:

- Exercise 9 (page S100) asks you to implement addition, subtraction, multiplication, and division on Peano numerals. It illustrates beautifully the ease with which an axiomatic specification can be implemented in Prolog.

- Exercises 25 and 26 (page S104) ask you to write an ask you to write an evaluator and type checker in Prolog. It's not worth doing both, but either illustrates how easy it is to take a formal operational semantics or a type system and implement it directly in Prolog—judgments in the the specification are expressed as predicates in the code.

- All the puzzle and game problems are entertaining, but the best of the lot is Exercise 34 (page S108), which asks you to solve a logic problem of Raymond Smullyan's. All these sorts of problems yield to a simple exhaustive search, but Exercise 34 can be solved using a more sophisticated strategy in which the code talks directly about what propositions imply what other propositions.

- Exercise 44 (page S112) asks you to extend μProlog by adding the cut. It showcases the ease with which continuation-passing style can be used to add a control operator.

D.10.1 Digging into the language

1. *Writing logic formally.* Using two clauses and a query, express Aristotle's famous syllogism (page S42) in Prolog.

2. *A predicate with multiple arities.* This exercise illustrates the use of the predicate `mother` at more than one arity.

   - For `mother/2`, proposition `mother(`M`, `C`)` should hold if person M is the mother of child C.
   - For `mother/1`, proposition `mother(`P`)` should hold if person P is a mother.

   The exercise has three parts:

   (a) Use your knowledge of family relationships to define one of these predicates in terms of the other.

   (b) The longest-reigning monarch in British history is Elizabeth II (a record likely to outlast this book). As I write, Elizabeth's eldest son and heir is Charles. Write whatever facts and rules of Prolog are needed to express their relationship. Use as few clauses as possible.

   (c) To verify that `mother(elizabeth)` is provable but `mother(charles)` is not, write unit tests.

3. *Simple logical reasoning.* Building on the previous exercise, suppose that a person celebrates Mother's Day if she *is* a mother or if he or she *has* a *living* mother.

   (a) Define a predicate `celebrates_md/1` that tells whether a person celebrates Mother's Day.

   (b) Define a predicate `living/1` that reflects current knowledge of the British royal family. Limit your attention to the reigning monarch and his or her descendants.

   (c) Define a relation `celebrants/2` such that `celebrants(`PS`, `CS`)` holds whenever list CS contains exactly those persons from PS who celebrate Mother's Day.

4. *Map coloring with a constraint.* The next few exercises build on the map-coloring examples in Section D.2. To start, get Prolog to produce a coloring of the British Isles map in which the Atlantic Ocean is colored blue.

5. *General-purpose predicates for maps.* In this exercise, you make it easier to define maps.

   (a) Define a predicate `alldifferent/2` predicate so that if C is a color and CS is a list of colors, `alldifferent(`C`, `CS`)` holds if and only if C is different from every color in CS.

   (b) Using the `alldifferent/2`, rewrite the rules for coloring the British Isles so that fewer premises are needed.

   (c) In an unlikely event of historic impact, France and Germany decide to unify to form one country, Europa—changing the map of Europe. Alter map (b) in Figure D.1 to reflect the new reality, by which I mean, write a Prolog program to color the new map. Use your `alldifferent` predicate.

   I regret the loss of the Iberian and Scandinavian peninsulas, not to mention southern Italy and eastern Europe, but ignore them.

6. *Four-coloring Western Europe.* The map of Western Europe, or at least that part shown in Figure D.1b, needs to be colored.

   (a) Add new clauses to the Prolog database so a map can be colored with *four* colors.

   (b) Write a Prolog program that colors the map in Figure D.1b. Ignore the Atlantic Ocean, the Iberian and Scandinavian peninsulas, and all the other interesting parts of Europe that aren't shown.

7. *Maps as data, not rules.* In Section D.2, each map is represented by an inference rule. But it is also possible to represent a map as data. For coloring, a good representation may involve an *adjacency list*. An adjacency list is a list of terms, each of which has the form `adj(`C`, `CS`)`, where C is associated with a country and each element of CS is associated with a country adjacent to C. For purposes of this problem, represent each country as a logical variable.

   I can represent a map by relating a list of countries to an adjacency list. As an example, a map of the island (not the country) of Ireland could be represented as follows:

   S100a. ⟨*exercise transcripts* S100a⟩≡ S100b ▷

   ```
   -> ireland([Atl, Ir, NI], [adj(Atl, [Ir, NI]), adj(Ir, [NI])]).
   ```

   (a) Using the adjacency-list representation, define predicate `coloring/1`, which is holds if its argument is a properly colored adjacency list. Consider using the predicate `alldifferent/2` from Exercise 5.

   S100b. ⟨*exercise transcripts* S100a⟩+≡ ◁ S100a S101a ▷

   ```
   -> [query].
   ?- ireland([Atl, Ir, NI], Rows), coloring(Rows).
   Atl = yellow
   Ir = blue
   NI = red
   Rows = [adj(yellow, [blue, red]), adj(blue, [red])]
   yes
   ```

   (b) Using the adjacency-list representation, color the full map of the British Isles.

8. *Understanding Prolog operations.* Give a step-by-step account of the rest of the computation for the coloring of the map of the British Isles, the first 13 steps of which are shown starting on page S68. I recommend against trying to simulate the computation by hand; instead, instrument the `britmap_coloring` rule with `print` predicates. Use the results to write your explanation.

9. *Peano arithmetic.* One of the great mathematical achievements of the nineteenth century was a logical theory of arithmetic. The simplest arithmetical theory is the theory of the natural numbers, which can be represented using the atom `zero` and the functor `succ`. For example, the term `succ(succ(succ(zero)))` represents the number 3. This representation is called a *Peano numeral,* after the mathematician who used these numerals to develop an axiomatic description of arithmetic, expressed in mathematical logic. Using Peano numerals, define these predicates:

   (a) Predicate `equals/2` tells if two Peano numerals are equal.

   (b) Predicate `plus/3` computes the sum of two Peano numerals.

(c) Predicate `minus/3` computes the difference of two Peano numerals. It succeeds only if the difference is representable as a Peano numeral—that is, if it is nonnegative.

(d) Predicate `times/3` computes the product of two Peano numerals.

(e) Predicate `div/4` divides one Peano numeral by another, computing the quotient and the remainder. If asked to divide by zero, `div` should fail, not loop forever.

(f) Predicate `print_peano/1` succeeds if its argument is a Peano numeral, and as a side effect, it prints the corresponding integer:

S101a. ⟨*exercise transcripts* S100a⟩+≡ ◁S100b S101b▷

```
?- print_int(succ(succ(zero))).
2
yes
```

Except for part (f), don't use the primitive `is` predicate.

10. *Boolean satisfaction.* A Boolean formula is a term in the following form:

- Any logical variable is a formula.
- `true` and `false` are formulas.
- If f is a formula, the term `not(`f`)` is a formula.
- If f_1 and f_2 are formulas, the term `and(`f_1`, `f_2`)` is a formula.
- If f_1 and f_2 are formulas, the term `or(`f_1`, `f_2`)` is a formula.

Write clauses for a Prolog predicate `satisfied` such that if f is a formula, the query `satisfied(`f`)` succeeds if and only if there is an assignment to f's variables such that f is satisfied. Issuing the query should also produce the assignment.

S101b. ⟨*exercise transcripts* S100a⟩+≡ ◁S101a S103▷

```
?- satisfied(and(A, and(B, not(C)))).
A = true
B = true
C = false
yes
```

11. *Horn clauses.* In this exercise, you write Prolog code to convert a Prolog clause into a Horn clause. There are a lot of definitions.

A *literal* is one of the following:

- An atom, which is called a *positive literal*
- A term of the form `not(`a`)`, where a is an atom, and which is called a *negative literal*

A *formula* is one of the following:

- A literal
- A term of the form `not(`f`)`, where f is a formula
- A term of the form `and(`f_1`, `f_2`)`, where f_1 and f_2 are formulas
- A term of the form `or(`f_1`, `f_2`)`, where f_1 and f_2 are formulas

A *Prolog clause* is a term of the form `(`a_0 `:-` $a_1, \ldots, a_n$`)`, where each a_i is an atom.

A *disjunction* is one of the following:

- A literal
- A formula of the form `or(`d_1`, `d_2`)`, where d_1 and d_2 are disjunctions

A *Horn clause* is a disjunction that contains at most one positive literal.

Write a Prolog predicate `is_horn/2` that converts between Prolog clauses and Horn clauses. It should run both forward and backward.

12. *Removing elements from a list.* The chapter defines `member`, which says if a list contains an element. To remove all copies of an element from a list, define predicate `stripped/3`, where `stripped(`XS`, `X`, `YS`)` holds whenever YS is the list obtained by removing all copies of X from XS.

13. *Splitting lists.* To split a list into equal or approximately equal parts, define and use these predicates:

    (a) Define `bigger/2`, where `bigger(`XS`, `YS`)` holds if and only if XS is a list containing more elements than YS.

    (b) Write a query that uses `bigger/2` and `appended/3` to split a list into two sublists of nearly equal lengths.

    (c) Write a query that uses `bigger/2` and `appended/3` to split a list into two sublists whose lengths differ by at most 1.

    (d) To help you write unit tests for your work, define `has_length/2`, where `has_length(`XS`, `N`)` holds if and only if XS is a list of N elements. If XS is a logical variable, or if any tail of XS is a logical variable, the resulting proposition need not be provable. In other words, if somebody hands you an N, don't try to conjure a suitable XS.

14. *Conversions between S-expressions and lists.* For purposes of this exercise, let us say that an S-expression is an atom, a number, or a list of zero or more S-expressions.

    (a) Define `flattened/2`, such that `flattened(`SX`, `AS`)` holds whenever SX is an S-expression and AS is a list containing the same atoms as SX, in the same order. The problem is analogous to the Scheme `flatten` function described in Exercise 8(d) on page 180.

    (b) For any list AS, there is an unbounded number of S-expressions SX such that `flattened(`SX`, `AS`)`. The issue is that SX may contain any number of empty lists, none of which contributes anything to AS.

       Address this issue by decomposing `flattened/2` into two or more predicates, one of which removes all empty lists, and the other of which flattens the result. Make sure the second predicate can be run backward.

    (c) A list of lists XSS is *triangular* if the first element of XSS has length 1, the second element has length 2, and so on. Define a suitable predicate `triangular/1`, which holds if its argument is triangular. Any auxiliary predicates you use should also be called `triangular`, but they may have a different arity.

    (d) Using your predicate from part (b) to generate candidates, and using `triangular` to test them, write a query that produces a triangular list containing the elements 1 to 6.

15. *Insertion sort.* Implement insertion sort by defining predicate `isorted/2`, where relation `isorted(`NS`, `MS`)` holds whenever MS is the result of sorting the list of numbers NS.

16. *Merge sort.* Implement merge sort by defining predicate `msorted/2`, where relation `msorted(`NS`, `MS`)` holds whenever MS is the result of sorting the list of numbers NS.

17. *Difference lists.* Program the following operations on difference lists. Don't simply transform them to ordinary lists.

    (a) `diffsnocced`

    (b) `diffreversed`

    (c) `diffquicksorted`

18. *Forward and backward.* These problems relate to the predicate `power`:

    (a) Under exactly what circumstances will `power` work in the backward direction?

    (b) Explain why the version of `power` in ⟨*bad version of* `power` S74d⟩ doesn't work.

19. *Forward and backward, revisited.* Explain why `quicksorted` can't be run backward.

20. *Termination.* Consider the definition of the predicate `fac` in chunk S74e. Do queries involving `fac` always terminate? If so, prove termination. If not, give an example query that fails to terminate, explain the problem, and show how to correct it.

21. *The blocks world.*

    (a) Change `transform` so that a move generated by `good_move` is rejected if it moves a block that has just been moved. Confirm that `transform` does not generate any plans that involve moving the same block twice in a row.

    (b) Change the representation of states to `state(`a`, `b`, `c`)`, where a is the location of block a b is the location of block `b`, and so on. Modify the program accordingly. Explain which representation you prefer, and why.

    (c) Instrument the code to measure how much backtracking is done by `transform/4`. In particular, count the number of moves generated by `good_move`. What is the ratio of that count to the number of moves in the solution?

       Measure the same ratio for `transforms2/4`. Does the superior answer produced by `transforms2` come at the cost of more backtracking?

22. *Visualizing backtracking.* The primitive predicate `print` prints a term when solved, but does nothing during backtracking. Create a predicate `backprint` which does nothing when solved, but which prints a term during backtracking. Perhaps surprisingly, `backprint` does not need to be a primitive predicate; you can write it in Prolog. Together, `print` and `backprint` make a crude tracing mechanism.

    S103. ⟨*exercise transcripts* S100a⟩+≡ ◁S101b S104b▷

    ```
    ?- member(X, [1, 2, 3]), print(trying(x, X)), backprint(failed(x, X)),
       member(Y, [3, 2, 1]), print(trying(y, Y)), backprint(failed(y, Y)),
       X > Y.
    trying(x, 1)
    trying(y, 3)
    ```

```
failed(y, 3)
trying(y, 2)
failed(y, 2)
trying(y, 1)
failed(y, 1)
failed(x, 1)
trying(x, 2)
trying(y, 3)
failed(y, 3)
trying(y, 2)
failed(y, 2)
trying(y, 1)
X = 2
Y = 1
yes
```

23. *Understanding the cut.* The cut is different from ordinary backtracking. Write rules for two Prolog predicates that behave differently and that are identical except that one uses a cut and one doesn't. Show a query that illustrates the difference between the two predicates.

24. *Detecting inequality using the cut.* Rewrite the predicate `not_equal` from Section D.8.3 (page S92) so that it still uses the cut, but it does not require the auxiliary predicate `equal`.

25. *Implementing an evaluator using operational semantics.* Throughout this book, we express operational semantics using inference rules. Since inference rules can be expressed directly in Prolog, we can easily write an interpreter based directly on the semantics. For example, consider these rules from the semantics of nano-ML:

$$\frac{}{\langle \mathrm{VAL}(v), \rho\rangle \Downarrow v} \qquad \text{(CONSTANT)}$$

$$\frac{\langle e_1, \rho\rangle \Downarrow v_1 \qquad v_1 = \mathrm{BOOLV}(\texttt{\#t}) \qquad \langle e_2, \rho\rangle \Downarrow v_2}{\langle \mathrm{IF}(e_1, e_2, e_3), \rho\rangle \Downarrow v_2} \qquad \text{(IFTRUE)}$$

Let's represent judgment $\langle e, \rho\rangle \Downarrow v$ as the Prolog predicate `eval(`e`, `ρ`, `v`)`. Then we can write these rules:

S104a. ⟨*sample rules for nano-ML evaluation* S104a⟩≡

```
eval(val(V), Rho, V).
eval(if(E1, E2, E3), Rho, V) :- eval(E1, Rho, true), eval(E2, Rho, V).
```

Write a complete set of rules of `eval` so that it forms an interpreter for nano-ML.

S104b. ⟨*exercise transcripts* S100a⟩+≡ ◁S103

```
?- eval(apply(val(plus), [val(2), val(2)]), [], V).
V = 4
yes
?- eval(apply(lambda([x], apply(val(plus), [var(x), var(x)])), [val(3)]), [], V).
V = 6
yes
```

26. *Implementing a type checker.* In Prolog, write a type checker for a simplified version of Typed μScheme in which both `lambda` and `type-lambda` take exactly one argument.

(a) Define a predicate `has_type(Gamma, Term, Type)` that holds when term `Term` has type `Type` in environment `Gamma`. You supply the environment and the term; Prolog computes the type. For the simplest possible type system, a checker in Prolog should take about a dozen lines of code.

(b) Add sums and products with `pair`, `fst`, `snd`, `inLeft`, `inRight`, and `either`.

(c) Add polymorphism.

Adding sums, products, and polymorphism will more than double part (a).

Here's a sample from my code:

S105. ⟨*sample run of a type checker in Prolog* S105⟩≡

```
| ?- has_type([],
       tylambda(alpha, tylambda(beta,
           lambda(p, cross(alpha, beta), pair(snd(var(p)), fst(var(p)))))), T).

T = forall(alpha,forall(beta,arrow(cross(alpha,beta),cross(beta,alpha))))
```

(d) Can you "run it backward" and get the engine to exhibit a term with a particular type? If not, why not?

(e) Can you modify your code to produce a derivation as well as a type? If not, why not?

D.10.2 Puzzles and games

Peg solitaire

The game "peg solitaire" is played on a board of ten holes arranged in a triangle:

```
   _
  o o
 o o o
o o o o
```

where _ represents an empty hole and `o` represents a hole with a peg in it. A "move" results when one peg jumps over another to land in a hole. The two pegs and hole must be colinear, and the stationary peg that was jumped over is removed from the board. So after a legal first move of the 1st peg on the third row (peg 4) we have:

```
   o
  _ o
 _ o o
o o o o
```

and after moving the last peg on the same row (peg 6) we have:

```
   o
  _ o
 o _ _
o o o o
```

and so on. When no peg can jump over any adjacent peg to land in a hole, the game is over. The object of the game is to leave a single peg, preferably in a designated hole. After my first attempt, I left this configuration:

```
   _
  o o
 _ _ _
_ _ _ o
```

If you want to play the game yourself, try it with small coins.

For the exercises below, number the pegs from 1, i.e., number the 10-hole layout like this:

```
   1
  2 3
 4 5 6
7 8 9 10
```

Solve the following problems:

27. *Knowing what 10-hole outcomes are possible.* Write Prolog rules such that the query `cansolve10(n)` succeeds if and only if 10-hole peg solitaire has a solution leaving `n` or fewer pegs. You can assume that `n` will always be passed in, e.g., we should expect `cansolve10(3)` to succeed always.

28. *Finding the best outcome with 10 holes.* Add new rules for `minleaving10` such that querying `minleaving10(N)` puts in `N` the minimum number of pegs that can be left on the board.

    Hint: use the cut.

For the next exercises, switch to a 15-hole layout:

or

```
    _
   o o
  o o o
 o o o o
o o o o o
```

29. *Knowing what 15-hole outcomes are possible.* Define predicate `minleaving` such that querying `minleaving(N)` puts in `N` the minimum number of pegs that can be left on the 15-hole board (like Exercise 28, but with 15 holes).

30. *Find a solution for any final peg.* Number the holes from top to bottom, left to right, and write Prolog rules such that `solution(n, M)` either produces in `M` a list of moves leaving a single peg in hole `n`, or fails if there is no such sequence. Represent a single move by the term `move(Start, Finish)`, so for example the two possible initial moves would be represented as `move(4,1)` and `move(6,1)`.

31. *Generalizing the initial configuration.* We don't always have to start with the top hole empty. Write Prolog rules such that `moves(S, F, M)` produces a sequences of moves `M` that takes the board from a configuration in which all holes *except* `S` have pegs to a configuration in which only hole `F` has a peg. Using these rules,

    (a) Write a query that finds a single location in which you can put an initial hole in order to make it possible to leave a single peg in hole 5.

(b) Time how long it takes to answer this query.

(c) Explain how you would speed it up.

Hints:

- Just as in the blocks-world example, think about a predicate that means "move `M` takes the board from configuration `B` to configuration `BB`."
- It might be easier to solve Exercise 32 and treat the problems above as special cases.
- The board has a symmetry group composed of threefold rotational symmetry plus reflection symmetry.

32. *Generalizing the number of pegs.* Solve one or more of Exercises 29 to 31, but make the number of holes in the triangle a parameter to the problem. For example, solve the board in the introduction by `solution(4, 1, M)` where 4 is the number of holes along one side of the triangle, 1 is the desired final hole, and `M` is the desired sequence of moves. Measure the performance cost of this generalization.

    Hint: The tough part is figuring out what's the numbering for a potential move. Think about shearing the board to form a lower-triangular matrix. What are the rules then for the permissible directions of motion? You may find it useful to number by row and column instead of just numbering the individual holes.

Logic problems

Mathematically, a "logic problem" is one that presents an N-dimensional Cartesian product space, then defines a relation by a set of constraints. The idea is for the relation to contain exactly one N-tuple, and the problem is to find it. If this description seems terribly abstract to you, fear not. Read the problems below, and maybe you'll recognize the genre. Even if you don't, solving logic problems in Prolog is easy and fun.

33. *Food Fest.* Andy, Bill, Carl, Dave, and Eric go out together for five evening meals, Monday through Friday. Each hosts one meal, and the host picks the food. They have fish, pizza, steak, tacos, and Thai food. After their exploit, the following facts transpire:

    (a) Eric had to miss Friday's dinner (so he could not host it)

    (b) Carl was host on Wednesday

    (c) They ate Thai on Friday

    (d) Bill, who hates fish, was the first host

    (e) Dave chose a steakhouse, where they ate the night before they had pizza.

    Write a Prolog program and query that tells who hosted each night and what food he selected. A solution should take the form of a Prolog list like the following:

```
[hosted(andy, fish, monday), hosted(bill, pizza, tuesday),
 hosted(carl, steak, wednesday), hosted(dave, tacos, thursday),
 hosted(eric, thai, friday)]
```

This example is not a solution: it doesn't fit facts (a), (d), and (e).

Notes: The classic way to solve this problem is "generate and test." You generate all possible solutions, then use the facts to rule out those that don't fit. But some care is needed; there are $5! \cdot 5! = 14,400$ possible solutions, and each solution has 120 possible representations, so if you're not careful you could wind up exploring over 1.7 million alternatives. If you're using a real Prolog system like XSB Prolog or SWI Prolog, this doesn't matter—these systems have so many optimizations that they find the first of the 120 possible representations in just a second or two. But if you're using μProlog, you need to cut down the search space.

- A good first step is to generate a single representation of the solution. Just pick a fixed order for either people, foods, or days. This step is worth taking even if you're using a real Prolog system; you'll get an answer ten times faster—essentially instantly.
- If you're using μProlog, you have to work harder. Apply the same idea we applied in the blocks world: change the generator so it generates only solutions that are consistent with known facts. In the *Food Fest* problem, try writing the potential solution not using a logical variable, but using a pattern that is consistent with what you know. For example, a potential solution might include the pattern

  ```
  hosted(carl, CFood, wednesday)
  ```

If you follow these two suggestions, you can get μProlog to produce an answer in under a second. If you try only the naïve generate-and-test strategy, μProlog can run for hours and consume gigabytes of RAM—without delivering a solution.

34. *The Stolen Jam.* The following logic problem is adapted from a problem by Raymond Smullyan, who has made a career out of this sort of nonsense.

> Someone has stolen the jam! The March Hare said he didn't do it (naturally!). The Mad Hatter proclaimed one of them (the Hare, the Hatter, or the Dormouse) stole the jam, but of course it wasn't the Hatter himself. When asked whether the Mad Hatter and March Hare spoke the truth, the Dormouse said that one of the three (including herself) must have stolen the jam.
>
> By employing the very expensive services of Dr. Himmelheber, the famous psychiatrist, we eventually learned that not both the Dormouse and the March Hare spoke the truth. Assuming, as one does, that fairy-tale characters either always lie or always tell the truth, it remains to discover who *really* stole the jam.

Write a Prolog program to discover who stole the jam. In particular, write rules for a predicate `stole/1` such that the query `stole(X)` succeeds if and only if `X` could have stolen the jam. The query should work even if `X` is left as a variable, in which case it should produce *all* the suspects who could possibly have stolen the jam. It is most likely that one of the three named characters is the culprit, but the culprit could be an outsider.

Hints:

- Like *Food Fest*, this problem can be tackled by exhaustive search of a large state space. The full state space for this problem should say who's lying, who's telling the truth, and of course who stole the jam.

- The most restricted possible state space has just one element: the identity of a suspect. This information could then be used to deduce who's lying and who's telling the truth.
- If you work only with simple predicates such as "the Hare is telling the truth" or "the Dormouse stole the jam," you may get stuck. Try such compound predicates as "if the Dormouse stole the jam, then the Hare is telling the truth."
- As mentioned on page S93, it's unwise to use the Prolog `not` predicate on anything except a ground term.
- Dr. Himmelheber is telling the truth.

35. *Murder, He Wrote.* This problem is by Teri Nutton; it was the Logic Problem of the Month in April, 1998.

> Five authors have just sent their latest murder stories to the publishers—so we all look forward to reading them soon. In the meantime, however, we intend to completely spoil your enjoyment of the novels, by inviting you to solve the problem of who murdered whom, as well as the motive involved and the location of the story!
>
> (a) Neither the butler nor the plumber committed the murder (which took place in Brighton) for the sake of an inheritance.
>
> (b) The revenge killing didn't take place in Fishguard or Dunoon. The artist didn't murder the partner (who was neither the victim killed in revenge nor the one murdered as the result of a power struggle).
>
> (c) The dentist murdered a cousin (but not for revenge or love) in Halifax.
>
> (d) The sister wasn't murdered in Brighton or Fishguard; and the victim in Fishguard wasn't the one killed for the love of someone. The butler didn't murder his partner.
>
> (e) In the novel in which the solicitor murders someone, the motive is power, but didn't involve the killing of a friend.

As in Exercise 33, write a Prolog program that says who killed whom, where, and for what motive.

D.10.3 Digging into the semantics

36. *Substitutions and functions.* Definition D.1 on page S57 defines a substitution. Prove these facts about substitutions:

    (a) Given a finite map $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$, show that this map determines a function from terms to terms, and prove that the function so determined has all the properties required of a substitution.

    (b) Given a function θ that maps terms to terms and that has all the properties required of a substitution, show that there exists some finite map $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ such that θ is the function determined by the map.

    (c) Prove that if θ_1 and θ_2 are substitutions, the composition $\theta_2 \circ \theta_1$ is also a substitution.

37. *Operational semantics.* Define a big-step operational semantics for Prolog, *without* the cut. The idea of such a semantics is that given a query, Prolog produces a *list* of substitutions which satisfy the query. In practice, the list is produced lazily, on demand, but your semantics can ignore this aspect.

Your semantics should be based on the judgment form $\boxed{D \vdash \theta s, gs}$, where D is a database, θs is a list of substitutions, and gs is a list of goals. The judgment says that given database D, query gs is satisfied by every substitution in θs. If θs is empty, the query cannot be satisfied. If θs is not empty, it contains all the solutions that Prolog finds, *in the order in which Prolog finds them.*

Your semantics should be able to express nontermination, but only weakly, like the semantics for Impcore: if Prolog's search does not terminate on a given D and gs, then there should be no derivation of $\boxed{D \vdash \theta s, gs}$. Your semantics need not be able to express whether Prolog might find *some* solutions *before* failing to terminate.

To express the search for clauses matching a goal, your semantics will need an auxiliary judgment $D, Cs \vdash \theta s, g :: gs$. This judgment is used only with a *nonempty* query of the form $g :: gs$. It says that the procedural interpretation finds substitutions θs that satisfy query $g :: gs$, given database D, and unifying g with the heads of clauses in Cs only.

To get you started, here are a few rules. The empty query is satisfied by the identity substitution.

$$\frac{}{D \vdash [I], [\,]} \qquad \text{(EMPTYQUERY)}$$

A nonempty query searches the entire database

$$\frac{D, D \vdash \theta s, g :: gs}{D \vdash \theta s, g :: gs} \qquad \text{(NONEMPTYQUERYSTART)}$$

If a goal does not unify with the (renamed) head of a clause, a property that I write $g \parallel G$, the search moves on to the next clause.

$$\frac{g \parallel G \qquad D, Cs \vdash \theta s, g :: gs}{D, (G \;\texttt{:-}\; Hs) :: Cs \vdash \theta s, g :: gs} \qquad \text{(WONTUNIFY)}$$

If there are no clauses left, the search doesn't produce any substitutions.

$$\frac{}{D, [\,] \vdash [\,], g :: gs} \qquad \text{(DATABASEEXHAUSTED)}$$

To write the remaining rule, which shows what happens when a goal *does* unify with the head of the next clause, you have to compute with multiple lists of substitutions. I recommend you use a powerful notation called *list comprehensions*, which have been popularized by the programming language Haskell. Here is an example of all pairs (x, y) where x is taken from xs and y is taken from ys:

$$[(x, y) \mid x \leftarrow xs, y \leftarrow ys].$$

In your rule, you are likely to take a list of substitutions $\theta' s$, and for each θ' in $\theta' s$, compute a second list of substitutions $\theta'' s$, and finally take the list of all the compositions. If $\theta'' s$ is related to θ' by relation $P(\theta', \theta'' s)$, you can write the list comprehension

$$[\theta'' \circ \theta' \mid \theta' \leftarrow \theta' s, P(\theta', \theta'' s), \theta'' \leftarrow \theta'' s].$$

Using this notation, write the last rule of the operational semantics for the procedural interpretation of Prolog. If you want to implement it, see Exercise 48 (page S114).

D.10.4 Digging into the interpreter

38. *Solving constraints.* Implement the constraint solver. That is, write function `solve` in chunk S79e. Given a constraint, `solve` should either return a substitution that satisfies the constraint, or raise the exception `Unsatisfiable`.

    This exercise is substantially the same exercise as Exercise 18 on page 448 of Chapter 7. If you need guidance, Chapter 7 explains constraint solving in detail.

39. *Understanding the occurs check.* Suppose you eliminate the occurs check. In this chapter, what examples go wrong? (You can instrument your solver to bark when the occurs check fails, or you can try another implementation of Prolog, which may have a flag that can be set to issue an error message when an occurs check fails.)

40. *Inequality as primitive.* Add a two-place primitive predicate `/=` (not equal).

    (a) Implement the basic version, which fails when applied to two identical integers or symbols and succeeds otherwise.

    (b) Implement the advanced version, which fails when applied to identical ground terms and succeeds otherwise.

    (c) Use either version in the blocks-world code, to replace the `different` predicate. Measure the difference in performance.

41. *Protecting primitive predicates.* Modify the μProlog interpreter so that if a user tries to define a clause in which the left-hand side is a built-in predicate, the interpreter issues an error message and refuses to add the clause to the database. For example, the following rule should cause an error:

    ```
    Z is X ^ N :- power(X, N, Z).
    ```

42. *Tracing flow through Byrd boxes.* Create a tracing version of the interpreter that logs every entry to and exit from a Byrd box. Use the following functions:

S111. ⟨*tracing functions* S111⟩≡ (S593b)

```
fun logSucc goal succ theta resume =
  ( app print ["SUCC: ", goalString goal, " becomes ",
               goalString (goalsubst theta goal), "\n"]
  ; succ theta resume
  )
fun logFail goal fail () =
  ( app print ["FAIL: ", goalString goal, "\n"]
  ; fail ()
  )
fun logResume goal resume () =
  ( app print ["REDO: ", goalString goal, "\n"]
  ; resume ()
  )
fun logSolve solve goal succ fail =
  ( app print ["START: ", goalString goal, "\n"]
  ; solve goal succ fail
  )
```

`goalString`	S597e
`goalsubst`	S596e

43. *Improving the database.* Every time it tries to satisfy a goal, our implementation of μProlog searches the *entire* database for matching clauses. More serious implementations use hash tables that are keyed on the *name* and *number of arguments* in the goal. Even without a hash table, one could cut down on searches by using

    ```
    type database = clause list env vector
    ```

    where element 0 of the vector contains 0-argument predicates, element 1 contains 1-argument predicates, and so on. Use either this data structure or some other one to change the implementation of the μProlog database, and measure the resulting speedups.

44. *Implementing the cut.* Add the cut to the μProlog interpreter.

    - Each Byrd box must take *three* continuations: $\kappa_{\texttt{succ}}$, $\kappa_{\texttt{fail}}$, and $\kappa_{\texttt{cut}}$. Supposing we are solving goal g_i based on the rule

      $$g \texttt{ :- } g_1, \ldots, g_n,$$

      the continuations play these roles:

      $\kappa_{\texttt{succ}}$ If we successfully satisfy $\theta(g_i)$, we pass θ to $\kappa_{\texttt{succ}}$. We also pass a resumption continuation so that if the solution of $g_{i+1}, \ldots, g_n$ fails, we can backtrack into g_i.

      $\kappa_{\texttt{fail}}$ If we fail to find a θ satisfying $\theta(g_i)$, we call $\kappa_{\texttt{fail}}()$, which is set up to backtrack to g_{i-1}.

      $\kappa_{\texttt{cut}}$ If g_i is a cut, we succeed and pass θ_{id} to $\kappa_{\texttt{succ}}$, but we *don't* pass a resumption continuation; if we backtrack into the cut, the entire goal g fails, *not* just g_i. Therefore the resumption continuation for $\kappa_{\texttt{succ}}$ must be the failure continuation for g.

    - In code chunk ⟨*search* **⟦prototype⟧** S80⟩, change the implementation of function `query` to add support for the cut. Functions `solveOne` and `solveMany` will both need an extra continuation argument $\kappa_{\texttt{cut}}$; the types of functions `search` and `query` should remain unchanged.

45. *Implementing* `not`. Add the primitive predicate `not` to the μProlog interpreter. You will not be able to do this simply using the existing mechanism for primitives, because implementing `not` requires a call to `solveOne`. Instead, treat `not` as a special case within `solveOne`.

46. *Preparing for reflective predicates.* In μProlog, the implementation of a primitive predicate has ML type

    $$\forall\alpha\,.\,\texttt{term list} \rightarrow (\texttt{subst} \rightarrow (\texttt{unit} \rightarrow \alpha) \rightarrow \alpha) \rightarrow (\texttt{unit} \rightarrow \alpha) \rightarrow \alpha.$$

    This type tells us that a Prolog primitive cannot affect the database. But primitives that affect the database, like `assert` and `retract`, are useful! In this exercise you change types in the interpreter so that primitive predicates become capable of reflection.

    (a) Change the type of every failure continuation from $\texttt{unit} \rightarrow \alpha$ to $\texttt{database} \rightarrow \alpha \times \texttt{database}$.

(b) Change the type of every success continuation from

$$\mathtt{subst} \to (\mathtt{unit} \to \alpha) \to \alpha$$

to

$$\mathtt{db} \to \mathtt{subst} \to (\mathtt{db} \to \alpha \times \mathtt{db}) \to \alpha \times \mathtt{db}.$$

where `db` is short for `database`.

(c) Change the type of `query` to

$$\begin{aligned}\forall\alpha\,.\,\mathtt{db} \to \mathtt{goal\ list} &\to (\mathtt{db} \to \mathtt{subst} \to (\mathtt{db} \to \alpha \times \mathtt{db}) \to \alpha \times \mathtt{db})\\ &\to (\mathtt{db} \to \alpha \times \mathtt{db}) \to \alpha \times \mathtt{db},\end{aligned}$$

where `db` is short for `database`.

(d) Change the type of every primitive predicate to

$$\begin{aligned}\forall\alpha\,.\,\mathtt{term\ list} \to (\mathtt{db} \to \mathtt{subst} \to (\mathtt{db} \to \alpha \times \mathtt{db}) \to \alpha \times \mathtt{db})\\ \to (\mathtt{db} \to \alpha \times \mathtt{db}) \to \alpha \times \mathtt{db},\end{aligned}$$

where `db` is short for `database`.

(e) Change function `process` in `processDef` to return the database computed by applying `snd` to the results of `query`. Pass `query` the failure continuation

```
(fn db => (print "no\n", db))
```

and the success continuation

```
(fn db => fn theta => fn resume =>
   if showAndContinue interactivity theta gs then resume db
   else (print "yes\n", db))
```

(f) Function `query` is also used to implement unit tests. Change the way `query` is called from `testIsGood`: give it success and failure continuations that are consistent with its new type.

(g) Using the new code, build and test μProlog.

47. *Implementing reflective predicates.* Using the interpreter from Exercise 46,

(a) Define primitive predicates `assert` and `retract` (page S94).

(b) Test your work by using `assert` to convert a map-coloring *adjacency list* (Exercise 7, page S100) into map-coloring *rules*. Color, yet again, the map of the British Isles.

(c) Test your work by using `assert` and `retract` to implement the general case of peg solitaire for a triangle of any size (Exercise 32, page S107).

To represent a fact, use a term. To represent a clause, wrap it in parentheses. As an example, μProlog parses the term

```
(sick(Patient) :- psychiatrist(Doctor), analyzes(Doctor, Patient))
```

as an application of functor `:-` to the two arguments `sick(Patient)` and `psychiatrist(Doctor)`. It then `analyzes(Doctor, Patient)`. The first argument represents the conclusion of the clause, and the remaining arguments represent the premises. This information should be enough to enable you to implement `assert` and `retract`.

48. *A definitional interpreter for μProlog.* Using your operational semantics from Exercise 37 (page S110), rewrite the core of the interpreter for μProlog. Here are some suggestions:

   - The main part of your rewrite should be a new function `solutions`, which takes a database and query and produces a `stream` of substitutions (Section H.5.2, page S228).
   - Function `solutions` should be specified by your operational semantics, which may include list comprehensions. To implement list comprehensions, I recommend a variation on `streamConcatMap`. I sometimes define

     S114a. ⟨*streams* S114a⟩≡ (S213a) S114b ▷

     ```
     every : 'a stream -> unit -> ('a -> 'b stream) -> 'b stream
     ```

     ```
     fun every xs () k = streamConcatMap k xs
     val run = ()
     ```

     Using `every` and `run`, the example list comprehension for the Cartesian product, $[(x, y) \mid x \leftarrow xs, y \leftarrow ys]$, is written as

     S114b. ⟨*streams* S114a⟩+≡ (S213a) ◁ S114a S114c ▷

     ```
     cartesian : 'a stream -> 'b stream -> ('a * 'b) stream
     ```

     ```
     fun cartesian xs ys =
       every xs run (fn x =>
         every ys run (fn y =>
           streamOfList [(x, y)]))
     ```

     This style lends itself to implementing list comprehensions.
   - Your `solutions` function should generate solutions for μProlog's primitive predicates, but the implementations of those predicates need not change. Those implementations expect success and failure continuations, which can be manipulated to produce a stream of substitutions. Use `streamOfCPS (p args)`, where `p` represents the primitive predicate, `args` represents its arguments, and `streamOfCPS` is defined as follows:

     S114c. ⟨*streams* S114a⟩+≡ (S213a) ◁ S114b S114d ▷

     ```
     fun streamOfCPS cpsSource =
       cpsSource (fn theta => fn resume => theta ::: resume ()) (fn () => EOS)
     ```
   - When `solutions` is complete, write a replacement `query` function that calls `cpsStream` on the result of `solutions`, where `cpsStream` is defined as follows:

     S114d. ⟨*streams* S114a⟩+≡ (S213a) ◁ S114c

     ```
     cpsStream : 'subst stream ->
     ('subst -> (unit->'a) -> 'a) -> (unit->'a) -> 'a
     ```

     ```
     fun cpsStream answers succ fail =
       case streamGet answers
         of NONE => fail ()
          | SOME (theta, answers) =>
              succ theta (fn () => cpsStream answers succ fail)
     ```

Part IV. Long programming examples

Appendix E contents

E

Extended programming examples

In the main text, most of the code examples have to be short. Some longer examples can be found below.

- Section E.1 (page S117) shows an evaluator for μScheme that is written in μScheme. Such an evaluator is called *metacircular*.

- As an example of using algebraic data types to implement a sophisticated data structure, Section E.2 (page S125) uses μML to demonstrate *2D-trees*: a geometric data structure that can quickly locate the point nearest a given coordinate.

- The very first applications of object-oriented programming languages were for simulation—it's right there in the name Simula. Section E.3 (page S137) demonstrates a discrete-event simulation in μSmalltalk.

E.1 Large μScheme example: A metacircular evaluator

One of the most intriguing features of Scheme is that programs are easily represented as S-expressions. By writing programs that manipulate such S-expressions, Scheme programmers can extend their programming environment more easily than with almost any other language. This extensibility accounts in part for the great power and variety of the programming environments in which Scheme and Lisp are often embedded (which, however, are beyond the scope of this book).

The treatment of programs as data was illustrated by McCarthy (1962) in a particularly neat way, namely by programming a "metacircular" interpreter for Lisp, that is, a Lisp interpreter written in Lisp. In this section, I follow McCarthy's lead, presenting a μScheme interpreter in μScheme. (I interpret just the core of μScheme, without the extended definitions, so there is no implementation of `use`, `check-expect`, `check-assert`, or `check-error`.)

I represent expressions exactly as if they were quoted literals. For example, I represent the expression `(+ x 4)` by the S-expression `'(+ x 4)`.

My evaluator is structured in much the way as the C version, but I use higher-order functions in ways that are not possible in C.

E.1.1 The environment and value store

I represent each location as a number. The store is an association list from numbers to values, so $\mathrm{dom}\,\sigma = \mathit{NUM}$. To support allocation, the store also maps the special key `next` to a fresh location n. The representation satisfies the invariant that $\forall i \geq n : i \notin \mathrm{dom}\,\sigma$.

S117. ⟨*eval.scm* S117⟩≡ S118a ▷

```
(val emptystore '((next 0)))
```

I make the store a global variable `sigma`.

S118a. ⟨*eval.scm* S117⟩+≡ ◁S117 S118b▷

```
⟨definition of find-c (from chunk 697b)⟩
(val sigma emptystore)
(define load  (l)   (find-c l sigma (lambda (x) x)
                            (lambda () (error (list2 'unbound-location: l)))))
(define store (l v) (begin (set sigma (bind l v sigma)) v))
```

To allocate, I use the special key `'next`. I give `allocate` the same interface as in C.

S118b. ⟨*eval.scm* S117⟩+≡ ◁S118a S118c▷

```
(define allocate (value)
  (let*
    ([loc (load 'next)])
    (begin
       (store 'next (+ loc 1))
       (store loc value)
       loc)))
```

Also as in C, `bindalloc` allocates a new location, stores a value in it, and returns that location. Similarly, `bindalloclist` allocates and initializes lists of locations.

S118c. ⟨*eval.scm* S117⟩+≡ ◁S118b S118d▷

```
(define bindalloc (name v env)
  (bind name (allocate v) env))
(define bindalloclist (xs vs env)
  (if (and (null? xs) (null? vs))
    env
    (bindalloclist (cdr xs) (cdr vs) (bindalloc (car xs) (car vs) env))))
```

By insisting that in the base case, both `xs` and `vs` must be empty, I ensure that if `xs` and `vs` have different lengths, the interpreter issues an error message and halts.

E.1.2 Representations of values

Within the metacircular interpreter, most values can be represented as themselves. That is, symbols can represent symbols, numbers can represent numbers, and so on. The exception is functions. A function is represented not as itself, but as a unary function that takes a list of arguments, possibly changes the store, and returns a single result. Such a function can be called a "function in list form."

A primitive μScheme function is transformed into list form by `apply-prim`. Function `apply-prim` knows that all primitives are either unary or binary.

S118d. ⟨*eval.scm* S117⟩+≡ ◁S118c S119a▷

```
(define apply-prim (prim)
  (lambda (args)
    (if (null? args)
      (error 'missing-arguments-to-primitive)
      (if (null? (cdr args))
        (prim (car args))
        (if (null? (cddr args))
          (prim (car args) (cadr args))
          (error (list2 'all-primitives-expect-one-or-two-arguments---got args)))))))
```

Function `apply-prim` makes no special effort to ensure that each primitive gets the right number of arguments. If an interpreter function applies `+` to only one argument, for example, I just get the underlying error message from the μScheme interpreter.

E.1.3 The initial environment and store

I can now build the initial environment. I start with an empty `env` and use `let*` to bind each primitive in sequence.

S119a. ⟨*eval.scm* S117⟩+≡ ◁S118d S119b▷

```
(define primenv ()
  (let*
      ([env '()]
       [env (bindalloc '+ (apply-prim +) env)]
       [env (bindalloc '- (apply-prim -) env)]
       [env (bindalloc '* (apply-prim *) env)]
       [env (bindalloc '/ (apply-prim /) env)]
       [env (bindalloc '< (apply-prim <) env)]
       [env (bindalloc '> (apply-prim >) env)]
       [env (bindalloc '= (apply-prim =) env)]
       [env (bindalloc 'car       (apply-prim car)        env)]
       [env (bindalloc 'cdr       (apply-prim cdr)        env)]
       [env (bindalloc 'cons      (apply-prim cons)       env)]
       [env (bindalloc 'println   (apply-prim println)    env)]
       [env (bindalloc 'print     (apply-prim print)      env)]
       [env (bindalloc 'printu    (apply-prim printu)     env)]
       [env (bindalloc 'error     (apply-prim error)      env)]
       [env (bindalloc 'boolean?  (apply-prim boolean?)   env)]
       [env (bindalloc 'null?     (apply-prim null?)      env)]
       [env (bindalloc 'number?   (apply-prim number?)    env)]
       [env (bindalloc 'symbol?   (apply-prim symbol?)    env)]
       [env (bindalloc 'function? (apply-prim function?)  env)]
       [env (bindalloc 'pair?     (apply-prim pair?)      env)])
    env))
```

E.1.4 The evaluator

I'm ready to explore the structure of the evaluator. Because the environment changes only when I make a function call, I define `eval` in curried form. It accepts an environment and returns a function from expressions to values. Just as in Chapter 5, I call this inner function `ev`.

S119b. ⟨*eval.scm* S117⟩+≡ ◁S119a S121g▷

```
⟨auxiliary functions for evaluation S120b⟩
(define eval (env)
   (letrec
       ([ev (lambda (e) ⟨result of evaluating expression e in environment env S120a⟩)]
        ⟨letrec bindings of functions used to evaluate abstract syntax S121c⟩)
     ev))
```

Each symbol stands for a variable, the location of which must be looked up in the environment. Other atoms evaluate to themselves.[1] Lists are function applications, unless they are abstract syntax.

S120a. ⟨*result of evaluating expression* `e` *in environment* `env` S120a⟩≡ (S119b)

```
(if (symbol? e)
  (load (find-variable e env))
  (if (atom? e)
    e
    (let ([first (car e)]
          [rest  (cdr e)])
      (if (exists? ((curry =) first) '(set if while lambda quote begin))
          ⟨evaluate first with rest as abstract syntax S120d⟩
          ⟨evaluate first to a function, and apply it to arguments from rest S120c⟩))))
```

To find a variable, I use `find-c`, so I can fail if the variable is not found.

S120b. ⟨*auxiliary functions for evaluation* S120b⟩≡ (S119b) S120e ▷

```
(define find-variable (x env)
  (find-c x env (lambda (x) x) (lambda () (error (list2 'unbound-variable: x)))))
```

Function application is straightforward. I don't bother to check to see if I am applying a non-function; the underlying μScheme interpreter does that for me. It takes much less space to write the code than to say what it does!

S120c. ⟨*evaluate* `first` *to a function, and apply it to arguments from* `rest` S120c⟩≡ (S120a)

```
((ev first) (map ev rest))
```

Abstract syntax is a bit more involved. I use brute force to check all the reserved words.

S120d. ⟨*evaluate* `first` *with* `rest` *as abstract syntax* S120d⟩≡ (S120a)

```
(if (= first 'set)    (binary  'set    meta-set    rest)
(if (= first 'if)     (trinary 'if     meta-if     rest)
(if (= first 'while)  (binary  'while  meta-while  rest)
(if (= first 'lambda) (binary  'lambda meta-lambda rest)
(if (= first 'quote)  (unary   'quote  meta-quote  rest)
(if (= first 'begin)  (meta-begin rest)
(error (list2 'this-cannot-happen---bad-ast first))))))))
```

The auxiliary functions `unary`, `binary`, and `trinary` unpack `rest` and check to be sure that it holds the correct number of elements. Function `holds-exactly` takes at most time proportional to `n`, no matter how long `xs` is.

S120e. ⟨*auxiliary functions for evaluation* S120b⟩+≡ (S119b) ◁ S120b S120f ▷

```
(define holds-exactly? (xs n)
  (if (= n 0)
    (null? xs)
    (if (null? xs)
      #f
      (holds-exactly? (cdr xs) (- n 1)))))
   (check-assert (holds-exactly? '(a b c) 3))
   (check-assert (not (holds-exactly? '(a b) 3)))
   (check-assert (not (holds-exactly? '(a b c d) 3)))
```

S120f. ⟨*auxiliary functions for evaluation* S120b⟩+≡ (S119b) ◁ S120e S121a ▷

```
(define unary (name f rest)
  (if (holds-exactly? rest 1)
    (f (car rest))
    (error (list3 name 'expression-needs-one-argument,-got rest))))
```

[1] The empty list shouldn't evaluate to itself; it should be an error, but I ignore that fine point.

S121a. ⟨*auxiliary functions for evaluation* S120b⟩+≡ (S119b) ◁S120f S121b▷

```
(define binary (name f rest)
  (if (holds-exactly? rest 2)
    (f (car rest) (cadr rest))
    (error (list3 name 'expression-needs-two-arguments,-got rest))))
```

S121b. ⟨*auxiliary functions for evaluation* S120b⟩+≡ (S119b) ◁S121a

```
(define trinary (name f rest)
  (if (holds-exactly? rest 3)
    (f (car rest) (cadr rest) (caddr rest))
    (error (list3 name 'expression-needs-three-arguments,-got rest))))
```

The `ast` functions themselves are straightforward, except for `lambda`. The easiest are `quote`, `if` and `while`.

S121c. ⟨`letrec` *bindings of functions used to evaluate abstract syntax* S121c⟩≡ (S119b) S121d▷

```
(meta-quote (lambda (e) e))
(meta-if    (lambda (e1 e2 e3) (if (ev e1) (ev e2) (ev e3))))
(meta-while (lambda (condition body) (while (ev condition) (ev body))))
```

A `set` expression requires us to find the location and rebind it.

S121d. ⟨`letrec` *bindings of functions used to evaluate abstract syntax* S121c⟩+≡ (S119b) ◁S121c S121e▷

```
(meta-set   (lambda (v e)
              (let ([loc (find-variable v env)])
                 (if (null? loc)
                    (error (list2 'set-unbound-variable v))
                    (store loc (ev e))))))
```

A `begin` expression evaluates arguments until it gets to the last. I use `foldl`.

S121e. ⟨`letrec` *bindings of functions used to evaluate abstract syntax* S121c⟩+≡ (S119b) ◁S121d S121f▷

```
(meta-begin (lambda (es) (foldl (lambda (e result) (ev e)) '() es)))
```

A `lambda` expression is the most fun. It must evaluate to a closure, so I use the real `lambda` to make a closure.

S121f. ⟨`letrec` *bindings of functions used to evaluate abstract syntax* S121c⟩+≡ (S119b) ◁S121e

```
(meta-lambda (lambda (formals body)
               (if (all? symbol? formals)
                 (lambda (actuals)
                   ((eval (bindalloclist formals actuals env)) body))
                 (error (list2 'lambda-with-bad-formals: formals)))))
```

E.1.5 Evaluating definitions

Evaluating a definition results in a new environment.

S121g. ⟨*eval.scm* S117⟩+≡ ◁S119b S122d▷

```
⟨functions used to evaluate definitions S122a⟩
(define evaldef (e env)
  (if (pair? e)
    (let ([first (car e)]
          [rest  (cdr e)])
      (if (= first 'val)
        (binary 'val (meta-val env) rest)
        (if (= first 'define)
            (trinary 'define (meta-define env) rest)
            (meta-exp e env))))
    (meta-exp e env)))
```

`all?`	ℬ 130
`atom?`	ℬ 103
`bindalloclist`	S118c
`caddr`	ℬ S319
`cadr`	ℬ 96
`car`	𝒫 162a
`cdr`	𝒫 162a
`curry`	ℬ 126
`env`	S119b
`error`	ℬ
`ev`	S119b
`eval`	S119b
`exists?`	ℬ 130
`find-c`	136
`foldl`	ℬ 131
`list2`	ℬ 96
`list3`	ℬ 96
`load`	S118a
`map`	ℬ 130
`meta-define`	S122b
`meta-exp`	S122c
`meta-val`	S122a
`not`	ℬ 639
`null?`	𝒫 162a
`pair?`	𝒫 162a
`store`	S118a
`symbol?`	𝒫 162a

The hardest definition to implement is `val`, which must see if the name `x` is already bound in the environment. I examine the environment using function `find-c` from Section 2.10 on page 136. If `x` is bound, I leave `env` alone; otherwise I extend `env` by binding `x` to the empty list. Once `x` is safely bound, I evaluate a `set` expression.

S122a. ⟨*functions used to evaluate definitions* S122a⟩≡ (S121g) S122b ▷

```
(define meta-val (env)
  (lambda (x e)
    (if (symbol? x)
        (let* ([env (find-c x env (lambda (_) env) (lambda () (bindalloc x '() env)))])
          (begin
            ((eval env) (list3 'set x e))
            env))
        (error (list2 'val-tried-to-bind-non-symbol x)))))
```

The `define` item is easy: I rewrite it into a `val` declaration.

S122b. ⟨*functions used to evaluate definitions* S122a⟩+≡ (S121g) ◁ S122a S122c ▷

```
(define meta-define (env)
  (lambda (name formals body)
    ((meta-val env) name (list3 'lambda formals body))))
```

Since I don't have a `read` primitive, I can't implement `use`. The only other "definition" is evaluation of a top-level expression.

S122c. ⟨*functions used to evaluate definitions* S122a⟩+≡ (S121g) ◁ S122b

```
(define meta-exp (e env)
  (begin
    (println ((eval env) e))
    env))
```

E.1.6 The `read-eval-print` *loop*

Function `read-eval-print` takes a list of definitions, evaluates each in turn, and returns the final environment and store.

S122d. ⟨*eval.scm* S117⟩+≡ ◁ S121g S122e ▷

```
(define read-eval-print (env es)
    (foldl evaldef env es))
```

Function `run` runs `read-eval-print` in an initial environment that contains just the primitives, then returns zero. (By returning the symbol `'done`, I make it possible to use `run` interactively without having to look at the final environment and store, which can be quite large.)

S122e. ⟨*eval.scm* S117⟩+≡ ◁ S122d

```
(define run (es)
  (begin (read-eval-print (primenv) es) 'done))
```

E.1.7 Tests

These tests exercise the functions `apply-prim`, `initialenv`, `meta-lambda`, `eval`, `evaldef`, `meta-if`, `meta-set`, `meta-val`, `meta-define`, `meta-exp`, `read-eval-print`, and `rep`.

S123a. ⟨*evaltest.scm* S123a⟩≡ S123b ▷

```
'(5 0 1 (Hello Dolly) 5 5 1 0)
(run
  '((define mod (m n) (- m (* n (/ m n))))
    (define gcd (m n) (if (= n 0) m (gcd n (mod m n))))
    (mod 5 10)
    (mod 10 5)
    (mod 3 2)
    (cons 'Hello (cons 'Dolly '()))
    (println (gcd 5 10))
    (gcd 17 12)))
```

These tests also exercise `meta-while` and `meta-begin`.

S123b. ⟨*evaltest.scm* S123a⟩+≡ ◁ S123a

```
'(5 0 1 #t 'blastoff 1 5 1 0)
(run
  '((define mod (m n) (- m (* n (/ m n))))
    (define not (x) (if x #f #t))
    (define != (x y) (not (= x y)))
    (define list6 (a b c d e f) (cons a (cons b (cons c (cons d (cons e (cons f '())))))))
    (define gcd (m n r)
      (begin
        (while (!= (set r (mod m n)) 0)
          (begin
            (set m n)
            (set n r)))
        n))
    (mod 5 10)
    (mod 10 5)
    (mod 3 2)
    (!= 2 3)
    (begin 5 4 3 2 1 'blastoff)
    (gcd 2 3 0)
    (gcd 5 10 0)
    (gcd 17 12 0)))
```

`bindalloc`	S118c
`error`	$\mathcal{B}$
`eval`	S119b
`evaldef`	S121g
`find-c`	136
`foldl`	$\mathcal{B}$ 131
`list2`	$\mathcal{B}$ 96
`list3`	$\mathcal{B}$ 96
`primenv`	S119a
`println`	$\mathcal{P}$
`symbol?`	$\mathcal{P}$ 162a

E.1.8 Exercises for the metacircular evaluator

The primary advantage of a metacircular evaluator is that it is easy to extend, so you can try out new language features. (It was once argued that a metacircular evaluator was a good way to write a language definition, but Reynolds [1972] found a flaw in that argument.) A significant disadvantage is that the metacircular evaluator may be slow, making it hard to try out your new features, especially if you want to run tests.

1. In the metacircular evaluator, the results of evaluating a top-level expression are not bound to `it`. Change the code in chunk S122c to correct this fault.

2. The metacircular evaluator doesn't implement any LET forms. Using syntactic sugar, as described in Sections 1.8 and 2.13, add those forms.

(a) As described in Section 2.13.1, add `let` to the metacircular evaluator using the law

$$\texttt{(let ([}x_1\; e_1\texttt{]} \cdots \texttt{[}x_n\; e_n\texttt{])}\; e\texttt{)} \;\equiv\; \texttt{((lambda (}x_1 \cdots x_n\texttt{)}\; e\texttt{)}\; e_1 \cdots e_n\texttt{)}.$$

You may find `map` more helpful than `foldr`.

(b) Similarly, add `let*` to the metacircular evaluator using the two laws

$$\begin{array}{lcl}
\texttt{(let* ()}\; e\texttt{)} & \equiv & e \\
\texttt{(let* ([}x_1\; e_1\texttt{]} \cdots \texttt{[}x_n\; e_n\texttt{])}\; e\texttt{)} & \equiv & \texttt{(let ([}x_1\; e_1\texttt{]) (let* ([}x_2\; e_2\texttt{]} \cdots \texttt{[}x_n\; e_n\texttt{])}\; e\texttt{))}
\end{array}$$

As usual, use the standard higher-order functions to help.

(c) Add `letrec` to the metacircular evaluator by rewriting

$$\texttt{(letrec ([x}_1\texttt{ e}_1\texttt{] ... [x}_n\texttt{ e}_n\texttt{]) e)}$$

to

$$\begin{array}{l}
\texttt{(let ([x}_1\texttt{ '()] ... [x}_n\texttt{ '()])} \\
\quad\texttt{(begin (set x}_1\texttt{ e}_1\texttt{) ... (set x}_n\texttt{ e}_n\texttt{) e))}
\end{array}$$

Use higher-order functions.

(d) With `let`, `let*`, and `letrec`, the evaluator should be powerful enough to evaluate itself. Measure how long the evaluator takes to evaluate itself evaluating `(+ 2 2)`.

3. Add short-circuit conditional primitives to the metacircular evaluator, using the syntactic sugar described in Section 2.13.3

(a) In full Scheme, `and` is variadic, and it works by *short-circuit evaluation,* like the `&&` operator from Section 2.13.3. This behavior can be expressed by the following laws:

$$\begin{array}{ll}
\texttt{(and)} & \equiv \texttt{\#t} \\
\texttt{(and p)} & \equiv \texttt{p} \\
\texttt{(and p}_1\texttt{ p}_2\texttt{ ... p}_n\texttt{)} & \equiv \texttt{(if p}_1\texttt{ (and p}_2\texttt{ ... p}_n\texttt{) \#f)}
\end{array}$$

Use these laws and `foldr` to add `and` to the metacircular evaluator in Section E.1.

(b) Similarly, use `foldr` to add variadic, short-circuit `or` to the metacircular evaluator, following these laws:

$$\begin{array}{ll}
\texttt{(or)} & \equiv \texttt{\#f} \\
\texttt{(or}\; e\texttt{)} & \equiv e \\
\texttt{(or}\; e_1 \cdots e_n\texttt{)} & \equiv \texttt{(let ([}x\; e_1\texttt{])} \\
 & \qquad \texttt{(if}\; x\; x\; \texttt{(or}\; e_2 \cdots e_n\texttt{)))}, \\
 & \qquad\quad \text{where } x \text{ does not appear in any } e_i
\end{array}$$

4. In many of my tests, the metacircular evaluator is annoyingly slow. This exercise suggests some improvements.

(a) Instead of making `next` an ordinary key in the store, represent the store as a pair `(cons next alist)`, so that you don't have to copy the store every time you allocate. Measure the effect on the speed of the metacircular evaluator, and measure the effect on the number of cells allocated by the underlying interpreter. (You will need to instrument `allocate` in chunk 162b.)

(b) Rewrite `bind` so that if a key does not appear in the association list, it conses a new key-attribute pair onto the front of the association list, without copying any existing pairs. Measure the effect on speed and allocation when running the metacircular evaluator.

(c) Rewrite `bind` to use move-to-front caching. That is, if association list `al2` is `(bind x y al)`, the list `(list2 x y)` should be the *first* element of `al2`, regardless of the position of `x` within `al`. This rewrite should also incorporate the improvement in part (b), so that if `x` is not bound in `al`, nothing is copied. Measure the effect on speed and allocation when running the metacircular evaluator.

(d) Measure the cumulative effect of the three preceding improvements on speed and allocation when running the metacircular evaluator.

Measure using the tests in chunks S123a and S123b.

E.2 Large μML example: 2D-trees

Algebraic data types are used to represent every expression and every definition in every interpreter from Chapter 5 onward. But algebraic data types are good for more than just interpreters—they are good representations of many data structures, especially those involving trees. In this section I present 2D-trees, which are used to look up geographic locations quickly.

E.2.1 Searching for points in 2D-trees

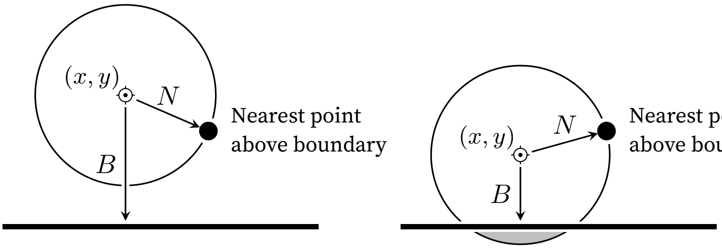
A 2D-tree is like a binary-search tree, but it is organized in two dimensions. The purposes of both trees are the same—search—but in a 2D-tree you are looking not for an exact match but for the point nearest a given location. With this background, here are some important differences:

- In a standard binary tree, each internal node contains a key, and each leaf is empty. In a 2D-tree, it's the other way around: an internal node contains only administrative information and subtrees, not any points—but each leaf contains a point.

- Order invariants are different. In a standard binary-search tree, keys are totally ordered. Values in the left subtree are smaller than the value at the root, and values in the right subtree are larger than the value at the root. Each subtree also obeys the order invariant.

  In a 2D-tree, keys are points in the plane, which can't be totally ordered. But each point has (x, y) coordinates, and any set of points can be totally ordered along either the x coordinate or the y coordinate, but not both. The order invariant depends on the administrative information at each internal node. At a *horizontal split*, the node contains the y coordinate of a horizontal boundary line, and two subtrees. The *below* subtree contains only points with smaller y coordinates than the horizontal line, and the *above* subtree contains only points with larger y coordinates than the horizontal line. At a *vertical split*, the boundary line is vertical, the root contains its x coordinate, and the *left* and *right* subtrees contain points with smaller and larger x coordinates, respectively.

  As an example, Figure E.2 (page S129) shows a 2D-tree that contains the locations and names of city halls near Boston, Massachusetts. Horizontal and vertical splits are shown by horizontal and vertical lines.



$N \leq B$: Needn't search below boundary $N > B$: Must search below boundary

Figure E.1: Search in a 2D-tree (the two important cases)

- When searching a standard binary-search tree, you're given a key and you search for exactly that key. If an internal node doesn't contain the key you're looking for, you go either to left or the right, and you look at just that subtree.

  When searching a 2D-tree, you're given an (x, y) coordinate pair, and you search for the point *nearest* to (x, y). In Figures E.1 and E.2, the search point (x, y) is depicted as a crosshair symbol ⊙. At an internal node, you still look left or right, up or down, but depending on what you find, you may have to look at *both* subtrees.

I hope you're already familiar with binary-search trees; you can implement some related codes in Exercise 14. This section explains 2D-trees: how search works, how to build one, and how they are used.

A search in a 2D-tree has only two nontrivial cases, both of which are shown in Figure E.1. The figure shows a single 2D-tree being searched at two different points; in each case, the search point (x, y) is shown as a crosshair ⊙. The tree being searched is a horizontal split, and the search point is above the boundary line. And in both cases, the nearest point in the *above* subtree (found by a recursive call) is the same. Also in both cases, the distance to that nearest point is N, and the distance to the boundary is B. Where the two cases differ is in whether they need to search below the boundary.

- On the left, $N < B$, which means the black dot is closer than the boundary line, and no point below the boundary can possibly be closer than the black dot. The search is over.

- On the right, $N > B$, so there might be a point in the shaded region, below the boundary, that is closer than the black dot. This case needs to search the *below* subtree.

The other interesting cases are obtained by rotating the diagram through angles of 90, 180, and 270 degrees. I want not to write the same code four times, so in each case I refer to the "near subtree" and "far subtree." The near subtree is the one that contains the search point, and the far subtree is the one that doesn't—the one on the far side of the boundary.

A 2D-tree is made up of `2Dpoints`, like the black dot in Figure E.1. Each point carries an x and y coordinate, plus a value of any type it likes.

S126. ⟨*gis.uml* S126⟩≡ S127a ▷

```
  (record ('a) 2Dpoint ([x : int] [y : int] [value : 'a]))
```

A value of type `(2Dtree` τ`)` is one of the following:

- A point `(POINT` p`)`, where p is a `(2Dpoint` τ`)`
- A horizontal split `(HORIZ` y *below above*`)`, where the y coordinate of every point in *below* is at most y, and the y coordinate of every point in *above* is at least y
- A vertical split `(VERT` x *left right*`)`, where the x coordinate of every point in *left* is at most x, and the x coordinate of every point in *right* is at least x

The structure and the types of all the parts, but not the ordering properties, are expressed using the algebraic data type `2Dtree`, which is defined as follows:

S127a. ⟨*gis.uml* S126⟩+≡ ◁S126 S127b▷

```
(implicit-data ('a) 2Dtree
   [POINT of (2Dpoint 'a)]
   [HORIZ of int (2Dtree 'a) (2Dtree 'a)] ; location below above
   [VERT  of int (2Dtree 'a) (2Dtree 'a)] ; location left right
)
```

To search a 2D-tree, I have to compare distances in the plane. But I don't want to *compute* distances—the computation includes a square root, and μML supports only integer arithmetic. Fortunately I can get the same results by comparing distances squared. Here is a function that gives the squared distance from (x, y) to a point.

S127b. ⟨*gis.uml* S126⟩+≡ ◁S127a S127c▷

```
point-distance-squared : (forall ['a] (int int (2Dpoint 'a) -> int))
```

```
(define square (n) (* n n))
(define point-distance-squared (x y p)
  (+ (square (- x (2Dpoint-x p)))
     (square (- y (2Dpoint-y p)))))
(check-expect (point-distance-squared 7 1 (make-2Dpoint 3 4 'test))
              25)
```

Before I tackle the search function, I want some auxiliary functions that embody the concepts of the search. For example, on the right of Figure E.1, if I have to search both sides of a boundary, I choose the closer of the two resulting points.

S127c. ⟨*gis.uml* S126⟩+≡ ◁S127b S128a▷

```
closer : (forall ['a] (int int (2Dpoint 'a) (2Dpoint 'a) -> (2Dpoint 'a)))
```

```
(define closer (x y p1 p2)
  (if (< (point-distance-squared x y p1) (point-distance-squared x y p2))
      p1
      p2))
```

Now I'm ready to start `nearest-point`. But there are nine cases! Luckily, one is trivial, and the other eight are all instances of Figure E.1. To handle the two cases shown in Figure E.1, I define auxiliary function `near-or-far` below. It takes x, y, the near subtree, the far subtree, and the distance squared B^2 between (x, y) and the boundary line. It returns the point closest to (x, y).

Using `near-or-far`, I define `nearest-point`. One case is `POINT`, four are from `HORIZ`, and four are from `VERT`. The two cases shown in Figure E.1 are from `HORIZ` where y is above the boundary; they are handled by the first call to `near-or-far`.

`2Dpoint-x`	$\mathcal{B}$
`2Dpoint-y`	$\mathcal{B}$

Each pair of other interesting cases (the rotations) is handled by a different call to `near-or-far`.

S128a. ⟨*gis.uml* S126⟩+≡ ◁S127c S128c▷

```
                nearest-point : (forall ['a] (int int (2Dtree 'a) -> (2Dpoint 'a)))
```

```
(define nearest-point (x y tree)
  (letrec (⟨definition of near-or-far within letrec S128b⟩)
    (case tree
      [(POINT p) p]
      [(HORIZ y-boundary below above)
           (if (> y y-boundary)
               (near-or-far x y above below (square (- y y-boundary)))
               (near-or-far x y below above (square (- y y-boundary))))]
      [(VERT  x-boundary left right)
           (if (> x x-boundary)
               (near-or-far x y right left  (square (- x x-boundary)))
               (near-or-far x y left  right (square (- x x-boundary))))])))
```

I define `near-or-far` in a `letrec` because μML hasn't got syntax for defining mutually recursive functions at top level.

Function `near-or-far` makes the decision in Figure E.1. The black dot is the closest point in the near subtree, at distance N from (x, y). If $N^2 \leq B^2$, I'm done; otherwise I search the far subtree and take the closer of the two points.

S128b. ⟨*definition of* `near-or-far` *within* `letrec` S128b⟩≡ (S128a)

```
[near-or-far
  (lambda (x y near far the-B-squared)
    (let* ([closest-near  (nearest-point x y near)]
           [the-N-squared (point-distance-squared x y closest-near)])
      (if (<= the-N-squared the-B-squared)
          closest-near ; don't need to search the far subtree
          (closer x y closest-near (nearest-point x y far)))))]
```

Now that I know how to search a 2D-tree, the next step is how to make one.

E.2.2 Making a balanced 2D tree

In typical applications, you build a 2D-tree for a fixed set of points, and you use it for a lot of searches. To make searches as fast as possible, you want the tree to be perfectly balanced, so the length of the path from the root to each leaf is the logarithm of the number of points. And to reduce the chances that you have to look across a boundary, the recommended heuristic is to alternate horizontal and vertical splits, hoping that alternating the directions of the boundaries will put them far away from the search point.

When I make a vertical split, how will I do it? I need to choose an x value such that half my points have smaller x's and half have larger x's. I sort points on their x coordinates, then split in the middle. To make a horizontal split, I do the same, but with y coordinates. For sorting, I define a higher-order function `sort-on`. When given a projection function, `sort-on` sorts a list of values using that projection. (I take `mergesort` as given.)

S128c. ⟨*gis.uml* S126⟩+≡ ◁S128a S129▷

```
mergesort : (forall ['a] (('a 'a -> order) -> ((list 'a) -> (list 'a))))
sort-on   : (forall ['a] (('a -> int) -> ((list 'a) -> (list 'a))))
```

```
⟨definition of mergesort (left as an exercise)⟩
(define sort-on (project)
  (mergesort (lambda (x1 x2) (Int.compare (project x1) (project x2)))))
```

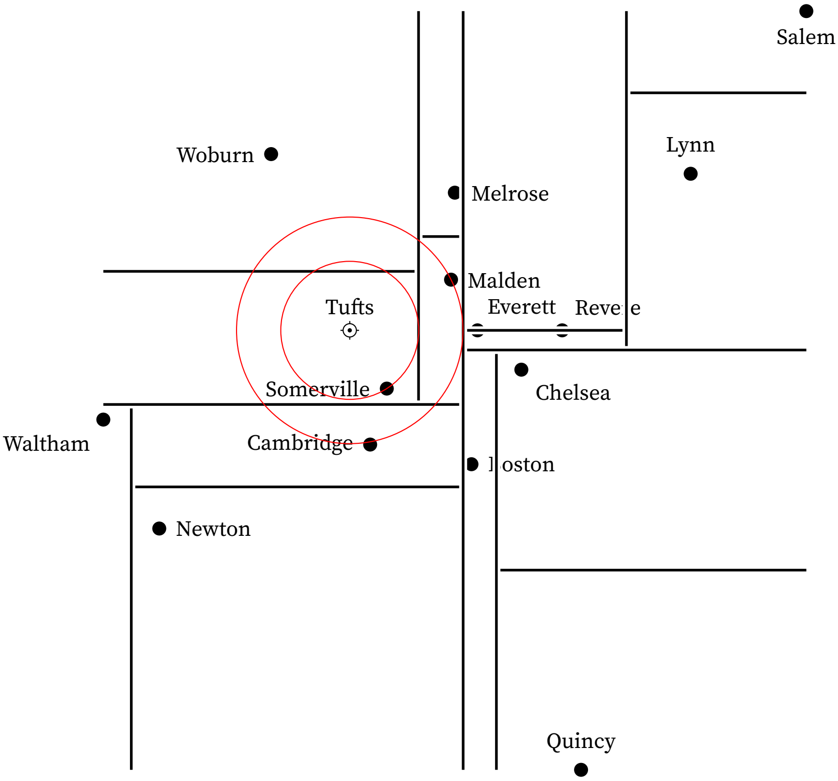


Figure E.2: Balanced 2D-tree of city halls near Boston, searched at Tufts (see page S134)

After sorting a list of points, I split it into halves. Function `halves` is specified as follows:

$$\texttt{(halves } xs\texttt{)} = \texttt{(pair } ys\ zs\texttt{)},$$
$$\text{where } xs = \texttt{(append } ys\ zs\texttt{)} \text{ and } |\texttt{(length } ys\texttt{)} - \texttt{(length } zs\texttt{)}| \leq 1.$$

S129. ⟨*gis.uml* S126⟩+≡ ◁S128c S130a▷

```
           halves : (forall ['a] ((list 'a) -> (pair (list 'a) (list 'a))))
  (check-expect (halves '(1 2 3 4))   (pair '(1 2) '(3 4)))
  (check-expect (halves '(1 2 3 4 5)) (pair '(1 2) '(3 4 5)))
```

Reasonable people would implement `halves` by using `length`, `take`, and `drop`. But I can't resist the opportunity to do it in one pass, as described on the next page.

`type 2Dtree`	S127a
`closer`	S127c
`HORIZ`	S127a
`Int.compare`	$\mathcal{B}$
`POINT`	S127a
`point-distance-squared`	S127b
`square`	S127b
`VERT`	S127a

I implement `halves` using a tail-recursive function that uses constant stack space. This function, `scan`, takes three parameters:

`left^`	A prefix of `xs`, reversed
`right`	Whatever part of `xs` is not in `left^`
`ys`	A list that is empty once `left^` contains half of `xs`

Getting `scan` right requires attention to loop invariants. But there's also a nice bit of pattern matching: function `scan` keeps going as long as `ys` has at least two elements; then it stops.

S130a. ⟨*gis.uml* S126⟩+≡ ◁ S129 S130b ▷

```
(define halves (xs)
  (letrec ([scan (lambda (left^ right ys)
                   ; invariants: xs = (revapp left^ right)
                   ;             (length xs) = (length ys) + 2 * (length left^)
                   (case ys
                     ((cons _ (cons _ zs))
                        (case right
                          ('() (error 'this-cannot-happen))
                          ((cons w ws)
                             (scan (cons w left^) ws zs))))
                     (_ (pair (reverse left^) right))))])
    (scan '() xs xs)))
```

Once I've split a list into halves, I draw a boundary between the largest small point (last element of the first list) and the smallest large point (first element of the second list). These elements are found by auxiliary functions `first` and `last`, which are defined only on nonempty lists.

S130b. ⟨*gis.uml* S126⟩+≡ ◁ S130a S130c ▷

```
(define first (xs) (car xs))
(define last (xs)
  (case xs
    [(cons x '()) x]
    [(cons _ ys)  (last ys)]
    ['()          (error 'last-of-empty-list)]))
```

Now that I can sort lists and split any list into two halves, I can build 2D-trees. As with search, I want to avoid duplicating code for the horizontal and vertical cases. To avoid duplicating code, I abstract over the coordinate. To abstract over X or Y, I need to know

- How to project the relevant coordinate

- How to make a split on that coordinate

I abstract these operations into a record of type `(forall ['a] (dimenfuns 'a))`.

S130c. ⟨*gis.uml* S126⟩+≡ ◁ S130b S130d ▷

```
(record ('a) coord-funs
     ([project  : ((2Dpoint 'a) -> int)                        ]
      [mk-split : (int (2Dtree 'a) (2Dtree 'a) -> (2Dtree 'a))]))
```

The x projection goes with the vertical split, and the y projection goes with the horizontal split.

S130d. ⟨*gis.uml* S126⟩+≡ ◁ S130c S131a ▷

```
vert-funs  : (forall ['a] (coord-funs 'a))
horiz-funs : (forall ['a] (coord-funs 'a))
```

```
(val vert-funs  (make-coord-funs 2Dpoint-x VERT))
(val horiz-funs (make-coord-funs 2Dpoint-y HORIZ))
```

When using vert-funs and horiz-funs, I want to alternate: vertical, horizontal, vertical, horizontal, and so on. But because I want you to generalize to more than two dimensions (Exercises 3 and 4), I code the alternation as follows: vertical; horizontal; start over; vertical; horizontal; start over; and so on. This idea generalizes to a sequence like "X, Y, Z, start over." To code it, I put the coordinates in a list all-coordinates, use the elements of that list until they are exhausted, then start again with all-coordinates. The coordinates not yet used are in list remaining-coordinates.

S131a. ⟨*gis.uml* S126⟩+≡ ◁S130d S131c▷

```
               2Dtree : (forall ['a] ((list (2Dpoint 'a)) -> (2Dtree 'a)))
(val all-coordinates (list2 vert-funs horiz-funs))
(define 2Dtree (points)
  (letrec
      ([build (lambda (points remaining-coordinates)
                  (case remaining-coordinates
                    ['() (build points all-coordinates)] ; start over
                    [(cons cfuns next-remaining)
                       (case points
                         [(cons pt '()) (POINT pt)]
                         [_   ⟨build tree using cfuns with points S131b⟩])]))])
      (build points all-coordinates)))
```

Given my coordinate functions, I extract projection and split-making functions, sort the points, split them into large and small halves, and compute the median coordinate for the split. The subtrees that go into the split are built using build with next-coords.

S131b. ⟨*build tree using* cfuns *with* points S131b⟩≡ (S131a)

```
(let* ([project    (coord-funs-project  cfuns)]
       [mk-split   (coord-funs-mk-split cfuns)]
       [sort       (sort-on project)]
       [points     (sort points)]
       [the-halves (halves points)]
       [small      (fst the-halves)]
       [large      (snd the-halves)]
       [_          (if (null? small) (error 'empty-small-tree) UNIT)]
       [_          (if (null? large) (error 'empty-large-tree) UNIT)]
       [average    (lambda (n m) (/ (+ n m) 2))]
       [median     (average (project (last small)) (project (first large)))]])
  (mk-split median (build small next-remaining) (build large next-remaining)))
```

As this stage I can write some rudimentary tests:

S131c. ⟨*gis.uml* S126⟩+≡ ◁S131a S132a▷

```
(val test-points
   (list3 (make-2Dpoint 10 12 'A)
          (make-2Dpoint  5  6 'B)
          (make-2Dpoint 33 99 'C)))
(val test-tree (2Dtree test-points))
(check-expect (2Dpoint-value (nearest-point  11  11 test-tree)) 'A)
(check-expect (2Dpoint-value (nearest-point 100 100 test-tree)) 'C)
```

For a more interesting test, I need more data.

E.2.3 Applying the 2D-tree: points of interest

The United States Geological Survey maintains a list of over two million *geographic names*, or as they are usually called by commercial GPS units, "points of interest."

The list is part of the U.S. Geographic Names Information System. Points of interest are partitioned into over 60 different "feature classes" ranging from Airport to Woods. In this section I use 2D-trees to find cities, towns, and city halls located near various points of interest in New England. The software that comes with this book includes lists of points of interest.

A geographic location is specified by its latitude and longitude. In the old days, these quantities were measured in degrees, minutes, and seconds or arc. Today, decimal degrees are widely used, and because μML provides only integers, I use millionths of a degree, also known as "microdegrees."

S132a. ⟨*gis.uml* S126⟩+≡ ◁S131c S132b▷

```
(record deg ([microdegrees : int]))
```

To compute the difference between two angles, I subtract their microdegrees.

S132b. ⟨*gis.uml* S126⟩+≡ ◁S132a S132c▷

```
deg-diff : (deg deg -> deg)
```

```
(define deg-diff (d1 d2)
  (make-deg (- (deg-microdegrees d1) (deg-microdegrees d2))))
```

A *point of interest* has a latitude, a longitude, and a name. Latitudes north of the equator are positive; latitudes south of the equator are negative. Longitudes east of Greenwich, England are positive; longitudes west of Greenwich, England are negative.

S132c. ⟨*gis.uml* S126⟩+≡ ◁S132b S132d▷

```
(record poi ([name : sym] [lat : deg] [lon : deg]))
```

Function `easy-poi` allows me to write the whole-number part and fractional part of latitude and longitude separately. This way I'm less likely to mess up the data entry.

S132d. ⟨*gis.uml* S126⟩+≡ ◁S132c S132e▷

```
easy-poi : (sym int int int int -> poi)
```

```
(define easy-poi (name lat-n lat-frac lon-n lon-frac)
  (let ([degrees (lambda (whole frac) (make-deg (+ (* 1000000 whole) frac)))])
    (make-poi name (degrees lat-n lat-frac) (degrees lon-n lon-frac))))
```

Am I ready to build a 2D-tree? Not yet. Microdegrees are accurate, but as x and y coordinates for a 2D-tree, they won't work, because of two problems:

- The closer we get to the Earth's poles, the closer together the lines of longitude are. 500 microdegrees of longitude represents a shorter distance than 500 microdegrees of latitude. My Euclidean calculations of distance squared would give wrong answers.

- If I square microdegrees, the resulting number won't be representable as a 32-bit integer. My calculations would cause machine arithmetic to overflow.

To address the distance-calculation problem, I approximate the Earth's surface as flat. The approximation is valid near a point, and the point I choose is the city of Boston, Massachusetts, whose inhabitants call it "the hub of the universe." Near Boston, there are 111,080 meters in a degree of latitude and 82,418 meters in a degree of longitude.

S132e. ⟨*gis.uml* S126⟩+≡ ◁S132d S132f▷

```
(val boston (easy-poi 'City-of-Boston 42 332221 -71 -016432))
(val meters-in-degree-lat 111080)
(val meters-in-degree-lon  82418)
```

To address the arithmetic-overflow problem, I compute distances not to the nearest meter, but to the nearest 30 meters.

S132f. ⟨*gis.uml* S126⟩+≡ ◁S132e S133a▷

```
(val distance-unit-in-meters 30)
```

I can now define functions that convert microdegrees into distances that make sense in a 2D-tree—as long as I stay aware of machine arithmetic. To convert microdegrees to meters, I could multiply by the number of meters in a degree, then divide by a million. But arithmetic would overflow. So instead of dividing *after* the multiplication, I divide each multiplicand by 1,000. And for better accuracy, I divide using function `/-round`, which rounds toward the nearest integer, and which is defined as follows:

S133a. ⟨*gis.uml* S126⟩+≡ ◁S132f S133b▷

```
(define /-round (dividend divisor)
  (/ (+ dividend (/ divisor 2)) divisor))
```

And finally, the conversion functions:

S133b. ⟨*gis.uml* S126⟩+≡ ◁S133a S133c▷

```
(define distance-of-microdegrees (meters-in-degree microdegrees)
  (let ([meters (* (/-round meters-in-degree 1000) (/-round microdegrees 1000))])
    (/-round meters distance-unit-in-meters)))

(define distance-of-degrees-lat (d)
  (distance-of-microdegrees meters-in-degree-lat (deg-microdegrees d)))
(define distance-of-degrees-lon (d)
  (distance-of-microdegrees meters-in-degree-lon (deg-microdegrees d)))
```

Using these functions, I can convert a point of interest into a proper `2Dpoint` whose x and y coordinates represent distance from Boston, as measured in units of `distance-unit-in-meters`.

S133c. ⟨*gis.uml* S126⟩+≡ ◁S133b S133d▷

```
2Dpoint-of-poi : (poi -> (2Dpoint poi))
```

```
(define 2Dpoint-of-poi (p)
  (let* ([delta-north (deg-diff (poi-lat p) (poi-lat boston))]
         [delta-east  (deg-diff (poi-lon p) (poi-lon boston))])
    (make-2Dpoint (distance-of-degrees-lon delta-east)
                  (distance-of-degrees-lat delta-north)
                  p)))
```

To simplify my examples, I define `nearest-to-poi`, which finds the point of interest nearest to some other point of interest.

S133d. ⟨*gis.uml* S126⟩+≡ ◁S133c S133e▷

```
nearest-to-poi : (forall ['a] (poi (2Dtree 'a) -> (2Dpoint 'a)))
```

```
(define nearest-to-poi (poi tree)
  (case (2Dpoint-of-poi poi)
    [(make-2Dpoint x y _) (nearest-point x y tree)]))
```

And here are some points of interest located in various New England states. Pinnacle Rock is a glacial erratic that offers a nice view of the city of Boston. The other points of interest listed here are all easily discoverable.

S133e. ⟨*gis.uml* S126⟩+≡ ◁S133d

```
(val pinnacle-rock    (easy-poi 'Pinnacle-Rock    42 439467 -71 -078238))
(val gillette-stadium (easy-poi 'Gillette-Stadium 42 090900 -71 -264300))
(val tufts            (easy-poi 'Tufts-University 42 408222 -71 -116402))
(val mt-washington    (easy-poi 'Mount-Washington 44 270500 -71 -303200))
(val the-breakers     (easy-poi 'The-Breakers     41 469722 -71 -298611))
(val mark-twain-house (easy-poi 'Mark-Twain-House 41 767139 -72 -700500))
```

`deg-microdegrees`	$\mathcal{B}$
`make-2Dpoint`	$\mathcal{B}$
`nearest-point`	S128a
`poi-lat`	$\mathcal{B}$
`poi-lon`	$\mathcal{B}$

Here is the search shown in Figure E.2 (page S129), except that it uses 83 city halls, not just the fourteen shown in the figure.

S134. ⟨*2D-trees transcript* S134⟩≡

```
-> (use gis.uml)
-> (use ne-city-halls.uml)
-> (val city-halls pois)
-> (val nearest-city-hall
      (let ([t (2Dtree (map 2Dpoint-of-poi city-halls))])
        (lambda (poi) (poi-name (2Dpoint-value (nearest-to-poi poi t))))))
nearest-city-hall : (poi -> sym)
-> (nearest-city-hall tufts)
Somerville-City-Hall/MA : sym
```

The city hall nearest Tufts is Somerville City Hall, but this search actually has to check four city halls (Figure E.2, page S129):

1. The first city hall searched is the one in the same region as Tufts: Somerville.

2. The boundary between Tufts and Woburn is closer than the Somerville City Hall, so the next point searched is across the boundary: Woburn City Hall. Somerville is closer.

3. The vertical boundary between the Woburn/Somerville subtree and the Melrose/Malden subtree is just barely closer to Tufts than Somerville City Hall is. So the code also searches east of that boundary.

4. Tufts is below the Melrose/Malden boundary, so it finds Malden. But if you extend that boundary line out to the west, you'll see Malden is further away from Tufts than the boundary is. So the code also looks above that boundary and finds Melrose. Malden is closer.

5. Finally, Somerville is closer than Malden. Therefore there's no need to look in the east half of the tree (the one containing Boston, Chelsea, Revere, Salem, and others).

My data set lists only 83 city halls, but the 2D-tree scales nicely to larger searches. This book is also accompanied by a data set of over 1500 cities and towns in New England. You can easily find that Gillette Stadium is nearest to Foxborough, The Breakers is nearest to Newport, and the Mark Twain House is nearest to Hartford. These queries are answered instantly. Building the 2D-tree takes a few seconds, if μML is built using the Moscow ML bytecode interpreter, or a quarter of a second, if μML is built using the MLton optimizing compiler. Faster if you bought your machine after 2015.

Task	*Time (milliseconds)*	
	Moscow ML	*MLton*
Infer types for code that builds list of `pois`	2,930	520
Convert 1527 `pois` to `2Dpoints`	350	220
Build 2D-tree	4,650	435
Find nearest city	1	1

Much time is also spent in type inference; the simple data structures used in Chapter 7 take time quadratic in the number of type variables. It is faster to store the point-of-interest data as S-expressions, read the S-expressions, and convert each S-expression to a `poi`.

E.2.4 Exercises based on 2D-trees

Geometrical search trees

The next group of exercises generalize the 2D-tree search code in Section E.2. You can implement other searches in two dimensions, the nearest-point search in higher dimensions, and a combination.

1. Generalize the code in Section E.2 to write a function `nearest-satisfying` that takes as arguments a search point (x, y), a predicate `p?`, and a 2D-tree t, and returns the nearest point whose value satisfies `p?`, if any.

   S135a. ⟨*exercise transcripts* S135a⟩≡ S135c ▷

   ```
   -> (check-type nearest-point-satisfying
         (forall ['a] (int int ('a -> bool) (2Dtree 'a) -> (option (2Dpoint 'a)))))
   ```

   S135b. ⟨*answers* S135b⟩≡ S135d ▷

   ```
   (use gis.uml)
   (val hello 'HELLO)
   (define nearest-point-satisfying (x y p? tree)
     (letrec (⟨definition of near-or-far-satisfying within letrec (from chunk 697b)⟩)
       (case tree
         ((POINT p) (if (p? (2Dpoint-value p)) (SOME p) NONE))
         ((HORIZ y-boundary below above)
              (if (> y y-boundary)
                  (near-or-far-satisfying above below (square (- y y-boundary)))
                  (near-or-far-satisfying below above (square (- y y-boundary)))))
         ((VERT  x-boundary left right)
              (if (> x x-boundary)
                  (near-or-far-satisfying right left  (square (- x x-boundary)))
                  (near-or-far-satisfying left  right (square (- x x-boundary))))))))
   ```

2. Generalize the code in Section E.2 to write a function `nearest-k-points`, which is like `nearest-point` except that it returns the nearest k points, where k is an additional parameter.

   S135c. ⟨*exercise transcripts* S135a⟩+≡ ◁ S135a S136a ▷

   ```
   -> (check-type nearest-k-points
         (forall ['a] (int int int (2Dtree 'a) -> (list (2Dpoint 'a)))))
   ```

   S135d. ⟨*answers* S135b⟩+≡ ◁ S135b

   ```
   (define nearest-k-points (x y k t)
      (case t
        ((POINT p) (list1 p))
        (_ (if (< k (+ x y)) '() '()))))
   ```

   As in the original search algorithm, don't look across a boundary unless you have to. Here are a few hints:

   - If you find points, return them in a list with the closest point first. Then when you have to look on both sides of a boundary, you can simply merge the two lists and return the first k elements of the merged list.
   - You might be asked for more points than you can supply. For example, if you reach a single `POINT` but are asked for a number $k > 0$, the best you can do is return a list containing just the one point you have.
   - If you're asked for the k nearest points, you can find up to k on the near side of the boundary, but on the far side of the boundary, you may not have to look for so many—depending on how many points you find on the near side, and where they are located, you might need only $k - 1$

points from the far side, or 3 points, or 0 points, or really any number from 0 to k inclusive.

- If you're asked for the nearest k points where $k = 0$, you don't have to look at anything; you just return an empty list.

3. In this exercise you generalize the 2D-tree to three dimensions. In the first parts of the exercise, you refactor the existing 2D-tree so that it still works in only two dimensions, but it is ready to be generalized:

   (a) Change the type of `nearest-point` to be

   ```
   (forall ['a] ((2Dpoint unit) (2Dtree 'a) -> (2Dpoint 'a)))
   ```

   (b) Introduce type `coordinate` using this definition:

   S136a. ⟨*exercise transcripts* S135a⟩+≡ ◁S135c S136b▷

   ```
   -> (implicit-data coordinate X Y)
   coordinate :: *
   X : coordinate
   Y : coordinate
   ```

   (c) Define function `project : (coordinate -> ((2Dpoint 'a) -> int))`.

   (d) Change the representation of 2D-tree so that there is only one value constructor for a split. To distinguish the vertical split from the horizontal split, arrange for the `SPLIT` value constructor to take a parameter of type `coordinate`:

   ```
   (implicit-data ('a) 2D-tree
      [POINT of (2Dpoint 'a)]
      [SPLIT of coordinate int (2Dtree 'a) (2Dtree 'a)])
   ```

   Now you can add the third dimension:

   (e) Change the representation of `2Dpoint` so that it includes a `z` coordinate.

   (f) Add new value constructor `Z` to type `coordinate`, and update function `project`.

   (g) Add a new record to the list `all-coordinates`. Change whatever else must change in functions `nearest-point` and `2Dtree` so they work with three dimensions.

4. In this exercise, you build on Exercise 3 to generalize the 2D-tree to arbitrarily many dimensions. Do Exercise 3 first, then complete the following parts.

   (a) Change the definition of `2Dpoint` so that a point stores a *list* of integer coordinates.

   (b) Define algebraic data type

   S136b. ⟨*exercise transcripts* S135a⟩+≡ ◁S136a

   ```
   -> (implicit-data coordinate [C of int])
   ```

   (c) Update function `project` so it uses the coordinate to index into the point's list of integers.

   (d) Update your nearest-point function to work with the new representations.

   (e) If you've completed Exercise 2, update your nearest-k-point function to work with the new representations.

Table E.3: Operations on histograms

Creators	
`new`	Returns a fresh histogram, distinct from any other, that maps every integer to a count of 0.
Observers	
`count-of`	`(count-of` i h`)` Returns the count associated with index i in histogram h.
`println`	Prints an attractive diagram of a range of entries in the histogram. The range includes all the indices that are associated with nonzero counts.
Mutators	
`inc`	Calling `(inc` i h`)` mutates h to increase by 1 the count associated with index i.
`inc-by`	Calling `(inc` i k h`)` mutates h to increase by k the count associated with index i.

(f) Define a function that given a number N and a list of N-dimensional points, builds a suitable search tree. A good place to start is with a list of `coord-funs` of length N.

If you complete this data structure, you can use it as part of a *k-nearest-neighbor classifier*. Such a classifier is a simple machine-learning tool, but still very effective on some problems, like classifying gestures based on a photograph of the human body. And as long as the number of dimensions is not too great, the search tree is reasonably efficient—it works well provided the number of points being searched is much larger than 2^N.

E.3 Extended μSmalltalk example: Discrete-event simulation

Object-oriented programming really shines when a lot of small objects, with simple methods, work together to create something powerful. To show the kind of interplay among classes and methods that characterizes well-designed object-oriented programs, I've put together example code that supports discrete-event simulation. This particular simulation explores a problem faced by my distinguished colleague Professor S.

Professor S's students are training robots to help urban search-and-rescue teams. For example, if firefighers cannot safely search a burning building, they might send one of Professor S's robots inside. Unfortunately, fireproof robots are madly expensive, so Professor S's lab has has only two robots, and Professor S's students have to take turns. To make sure every student gets a turn, Professor S wants to limit each student to at most t minutes on any given robot; after t minutes, another student gets a turn. How should Professor S choose t? Specifically, what value of t minimizes the time that the average student can expect to wait for a robot?

Professor S could experiment with different values of t in the robot lab, but the average waiting time is also affected by the number of students in the lab and by other conditions that are hard to reproduce, so it's not clear what the results would mean. And if some values of t are worse than others, the experiment is not fair to the students who are in the lab while those values are in force. The alternative I explore below is to write a program that *simulates* the lab—students arriving, waiting for robots, and using robots—and run the simulation multiple times with different values of t. Simulation has all sorts of advantages: it doesn't disrupt students; it's cheap enough to run many experiments; and the laboratory conditions are totally controlled and reproducible. But there's one huge caveat: the simulation might not model what would really happen. In this section, I don't worry about realism; I just try to demonstrate Smalltalk.

In the robot lab, each interesting event happens at a discrete point in time: a student arrives and wants a robot; a student actually gets to use a robot; or because t minutes have elapsed, a student has to relinquish a robot. This situation calls for a *discrete-event simulation*. Discrete-event simulations are used for many problems, including such problems as evaluating plans for handling baggage at an airport, estimating traffic flow over a highway, or deciding what inventory to keep in a warehouse.

Other kinds of simulation work with continuous variables, like the voltage of electrons in a circuit or the density of molecules in the atmosphere. These are *continuous-event simulations*, and the techniques used to implement them are very different from those I explore below.

E.3.1 Designing discrete-event simulations

Smalltalk's object-oriented style is a good fit for simulation. A full Smalltalk-80 system includes tools for modeling, viewing, and controlling simulations. Using these tools is so easy that even novice programmers can create interesting simulations. In this section, I draw on these tools to create a discrete-event simulation that highlights object-oriented programming techniques.

- If an entity in the system is allowed to take actions, like grabbing a robot, it is represented by a *simulation object*. In my example, each student is represented by a simulation object. A student takes such actions as asking for a robot or relinquishing a robot.

- If an entity represents a finite supply of some good or service—like a robot, a baggage cart, or a warehouse shelf—that entity is called a *resource*. The simulation classes that come with Smalltalk-80 provide special support that helps simulation objects acquire, release, or wait for resources. A resource might be represented by a single object, but it's also possible that a group of identical resources can be represented by a single object. An object representing a resource keeps track of the state of that resource as the simulation progresses. In my example, the only significant resource is the lab with its two fireproof robots.

- The overall simulation is orchestrated by an object, called "the simulation," whose class inherits from `Simulation`:

  - It keeps track of simulated time.
  - It schedules and runs every simulated event, always knowing what action is supposed to happen next.
  - It responds to requests for resources, and if a resource isn't available, it puts the requesting simulation object on a queue to wait.

`startUp`	Initialize the simulation, including scheduling at least one event.
`proceed`	Simulate the next event.
`finishUp`	End the simulation and save (or print) the results.
`enter: anObject`	Notify the receiver that a new object (the argument) has entered the simulation.
`exit: anObject`	Notify the receiver that the argument has left the simulation.
`time-now`	Answer the current simulated time
`scheduleEvent:at: anEvent aTime`	Schedule the event `anEvent` to occur at the given simulated time. The `anEvent` object must respond to the `takeAction` message, which is sent to it when the scheduled time arrives.
`scheduleEvent:after: anEvent aTimeInterval`	Schedule the event to occur after the given (simulated) time interval will have passed.
`scheduleRecurringEvents:using: aClass aStream`	Get a time interval from `aStream` by sending it the `next` message, then schedule a new, anonymous event to occur after that interval. When the new event occurs, create a new simulation object by sending message `new` to `aClass`, then repeat indefinitely. The effect is a series of recurring events at time intervals given by `aStream`.
resource methods	(Every subclass of *simulation* provides subclass-specific methods that are used to acquire and release simulated resources.)

(a) Instance protocol for `Simulation`

`ActiveSimulation`	Holds the value of the currently active `Simulation` object.

(b) Global variable used by `Simulation`

Figure E.4: Partial interface to class `Simulation`

- It keeps track of whatever information about the simulation is important, so when the simulation is over, it can report conclusions. In my example, the simulation tracks the amount of time students spend waiting for robots.

As you walk through the design and implementation of the robot-lab simulation, keep an eye out for two salient aspects of the object-oriented style: you will see methods, like the `Simulation` instance methods, which are intended to be easy to reuse; and you will also see that, unlike in procedural programming, the actions needed to implement an algorithm tend to be “smeared out” over multiple methods of multiple classes, making the algorithm a bit difficult to follow.

Figure E.4 sketches the protocol that I suggest for simulations. The protocol is adapted from similar protocols in the Smalltalk-80 blue book (Goldberg and Robson 1983):

- The first three methods of a `Simulation` instance make it possible to start, run, and end the simulation. A subclass typically adds extra initialization and finalization to the `startUp` and `finishUp` methods.
- The `enter:` and `exit:` methods allow a subclass to keep track of which "active" simulation objects are participating in the simulation.
- The `time-now` method and scheduling methods allow all participants to know the current time and to schedule future events.
- *Resource methods* are simulation-specific. They enable active objects to acquire and release resources, and they should be provided by a subclass of `Simulation`.
- Finally, the design assumes that only one simulation runs at a time. It is stored in global variable `ActiveSimulation`.[2]

S140. ⟨*simulation classes* S140⟩≡ S141a ▷

```
(val ActiveSimulation nil)
```

Using this design, you can expect most of a simulation's methods to send messages that fall into three categories:

- A message from a simulation object to the simulation. It notifies the simulation of entry or exit, requests or releases a resource, schedules an event, or asks about the current time.
- A message from the simulation to a simulation object. It grants access to a resource or tells the simulation object to act. Granting access is simulation-specific, but to tell a simulation object to act, every simulation sends the `takeAction` message. This message is the only message to which all simulation objects must respond.
- A simulation-specific message either from the simulation or from a simulation object to a resource or to another passive entity. It tells the receiver to change its state.

The rest of this section shows how to implement the `Simulation` class, how to implement a `RobotLabSimulation` subclass, and how to implement the simulation objects and resources that support the robot-lab simulation.

E.3.2 Implementing the `Simulation` *class*

The methods for scheduling and simulating events are common to all simulations and should therefore be implemented just once, in the `Simulation` class. Several methods are specialized by different subclasses, and simulation-specific resource methods are implemented only in subclasses.

To implement the protocol in Figure E.4, I need only two instance variables:

- Variable `now` holds the current simulated time. A simulation is free to use any representation of time that answers the `Magnitude` protocol in Figure 10.20 on page 650. (A simulation needs only to know which of two times is smaller, because the event with the smallest time is the one that occurs the soonest.)

[2] In Smalltalk-80, `ActiveSimulation` would be a class variable (page 704), not a global variable.

`isEmpty`	Answer `True` if and only if the receiver holds no events.
`at:put: aTime anEvent`	Add `anEvent` to the receiver, scheduling it to occur at time `aTime`.
`removeMin`	Provided the receiver is not empty, answer an `Association` in which the `value` is an event that is contained in the receiver and has minimal time, and the `key` is the associated time.

Figure E.5: Instance protocol for class `PriorityQueue`

- Variable `eventQueue` holds events that have not yet taken place, but are scheduled to occur in the simulated future. The event queue may also hold events that are scheduled to occur at time `now`.

S141a. ⟨*simulation classes* S140⟩+≡ ◁S140 S143b▷

```
(class Simulation
    [subclass-of Object]
    [ivars now eventQueue]
    (method time-now () now)
    ⟨more methods of class Simulation S141b⟩
)
```

The main invariant of a simulation is that at each point in time, the state of the objects in the simulation faithfully represents the state of the entities at the time stored in `now`. The states and the clock change only when there's an *event*. Events that are planned to occur in the simulated future are stored in `eventQueue`, which is a collection of events keyed by future time. The protocol for `eventQueue` is given in Figure E.5, and its implementation is discussed further in Exercise 1 on page S154.

Initializing, finalizing, and stepping a simulation

Initializing a simulation initializes the two instance variables and the global variable `ActiveSimulation`. To add initialization for its own private state, a subclass defines its own `startUp` method, which should send `(super startUp)`.

S141b. ⟨*more methods of class* `Simulation` S141b⟩≡ (S141a) S141c▷

```
(method startUp ()
    (set now 0)
    (set eventQueue (PriorityQueue new))
    ((ActiveSimulation isNil) ifFalse:
         {(self error: 'multiple-simulations-active-at-once)})
    (set ActiveSimulation self)
    self)
```

Finalizing the simulation resets `ActiveSimulation` to `nil`.

S141c. ⟨*more methods of class* `Simulation` S141b⟩+≡ (S141a) ◁S141b S141d▷

```
(method finishUp ()
    (set ActiveSimulation nil)
    self)
```

The `proceed` method simulates the next event in the queue.

S141d. ⟨*more methods of class* `Simulation` S141b⟩+≡ (S141a) ◁S141c S142a▷

```
(method proceed () [locals event]
   (set event (eventQueue removeMin))
   (set now (event key))
   ((event value) takeAction))
```

(This implementation is too simple-minded: it always sends `removeMin` to the `eventQueue` object, but the client object that sends `proceed` can't know if `removeMin` is safe. The `Simulation` protocol should be enriched so that clients can call `proceed` safely, as described in Exercise 7 on page S157.)

I define a method `runUntil:`, which runs events from the queue in order of increasing time until there are no more events—or until a time limit is reached. This is the method I use to run robot-lab simulations.

S142a. ⟨*more methods of class* `Simulation` S141b⟩+≡ (S141a) ◁ S141d S142b ▷

```
(method runUntil: (timelimit)
    (self startUp)
    ({(((eventQueue isEmpty) not) & (now <= timelimit))} whileTrue:
       {(self proceed)})
    (self finishUp)
    self)
```

Tracking entry and exit of simulation objects

In a general simulation, the `enter:` and `exit:` methods don't do anything. To know what needs to be done when a simulation object enters or exits the simulation, I need a simulation-specific method. Such a method would be defined on a subclass of `Simulation`, but because a subclass is not *required* to do anything on entry or exit, trivial implementations of `enter:` and `exit:` are provided here.

S142b. ⟨*more methods of class* `Simulation` S141b⟩+≡ (S141a) ◁ S142a S142c ▷

```
(method enter: (anObject) nil)
(method exit:  (anObject) nil)
```

Scheduling events

The fundamental scheduling operation is to schedule an event at a given time. An example would be to tell the simulation, "schedule the lab to open at 3:00PM." I schedule an event by using the `at:put:` method of class `PriorityQueue` to add the event to the event queue.

S142c. ⟨*more methods of class* `Simulation` S141b⟩+≡ (S141a) ◁ S142b S142d ▷

```
(method scheduleEvent:at: (anEvent aTime)
    (eventQueue at:put: aTime anEvent))
```

It's often convenient to schedule an event not at an *absolute* time, but at a time that is *relative* to the current time. An example would be "schedule this student to relinquish her robot at time t minutes from now."

S142d. ⟨*more methods of class* `Simulation` S141b⟩+≡ (S141a) ◁ S142c S143a ▷

```
(method scheduleEvent:after: (anEvent aTimeInterval)
    (self scheduleEvent:at: anEvent (now + aTimeInterval)))
```

The most interesting scheduling method is one that schedules *recurring* events. This method takes two arguments:

- An `eventFactory` provides an unlimited supply of events: to create a new event, send message `new` to the factory. An `eventFactory` is typically (but not always) a class.

- A `timeStream` provides a sequence of intervals that should elapse between events. The next interval is obtained by sending the message `next` to a `timeStream`. In a full Smalltalk-80 system, times in a stream are computed using a random-number generator. For example, "random arrival times" are normally modeled using a random-number generator that uses a Poisson distribution.

To implement recurring events, I define a new class of simulation object which is called `RecurringEvents`. An object of class `RecurringEvents` is initialized with an `eventFactory` and a `timeStream`.

S143a. ⟨*more methods of class* `Simulation` S141b⟩+≡ (S141a) ◁S142d

```
(method scheduleRecurringEvents:using: (eventFactory timeStream)
    ((RecurringEvents new:atNextTimeFrom: eventFactory timeStream)
     scheduleNextEvent))
```

An object of class `RecurringEvents` represents an infinite stream of future events. Every object in this class answers the `scheduleNextEvent` message, for which the protocol requires the receiver to remove the next event from itself and schedule it.

The implementation is subtle. When the object receives `scheduleNextEvent`, it pulls the next time from the `timeStream`, but it schedules *itself* as a proxy for the real event that is supposed to occur at the next time. Then, when the scheduled event occurs, the proxy receives the `takeAction` message, and it responds by using the factory to create the real event that is supposed to occur at this time. This implementation ensures that the `new` message is sent to a factory object at the appropriate simulated time. Finally, `takeAction` finishes by scheduling the *next* recurring event. All this action is easier to code than to explain: the two methods together need only 5 lines of μSmalltalk. The rest of the code is used only for initialization.

S143b. ⟨*simulation classes* S140⟩+≡ ◁S141a S144a▷

```
(class RecurringEvents [subclass-of Object]
    ; represents a stream of recurring events, each created from
    ; 'factory' and occurring at 'times'
    [ivars factory times]
    (method scheduleNextEvent ()
        (ActiveSimulation scheduleEvent:after: self (times next)))
    (method takeAction ()
        (factory new)
        (self scheduleNextEvent))
    (class-method new:atNextTimeFrom: (eventFactory timeStream)
        ((super new) init:with: eventFactory timeStream))
    (method init:with: (f s) ; private
        (set factory f)
        (set times s)
        self)
)
```

The initialization methods (the class method `new:atNextTimeFrom:` and the instance method `init:with:`) implement the common pattern, first shown in Section 10.1, in which I create an object by sending a message to a class method, which then uses an instance method to initialize the new object.

E.3.3 Implementing the robot-lab simulation

The implementation of a robot-lab simulation follows the plan sketched above:

- A single object of class `RobotLabSimulation` (a subclass of `Simulation`) orchestrates the simulation and keeps track of its state.
- Every simulation object that acts in the system is a student, each one of which is represented by an instance of class `Student`.
- The only resource I need to simulate is the lab itself, with its two robots. The lab is simulated by a single object of class `Lab`. The queue of students who are waiting to use the resource is maintained by the `RobotLabSimulation`.

The `Lab` class is the simplest, and I start there. Then `RobotLabSimulation`, and finally the most complex class, `Student`.

As you read the code, keep in mind the distinction between an event's being *scheduled* and that event's actually *occurring*. When an event is scheduled, it is simply added to the `eventQueue`; nothing else happens. Scheduling is the job of the `Simulation` superclass's scheduling methods. When an event *occurs* (when a simulation object receives `takeAction` or a factory object receives `new`), things happen, and the state of the simulation can change. Changing the state is the job of the `enter:` and `exit:` methods as well as the subclass-specific resource methods.

The class `Lab`

This class represents the state of the lab as a pair of Booleans, each of which says if a robot is available. Its protocol allows clients to check if there is a free robot (`hasARobot?`), get a robot (`takeARobot`), and give up a robot (`releaseRobot:`). All these methods are called when events occur, not when they are scheduled.

S144a. ⟨*simulation classes* S140⟩+≡ ◁ S143b S144b ▷

```
(class Lab
    [subclass-of Object]
    [ivars robot1free robot2free]
    (class-method new () ((super new) initLab))
    (method initLab () ; private
        (set robot1free true)
        (set robot2free true)
        self)
    (method hasARobot? () (robot1free | robot2free))
    (method takeARobot ()
         (robot1free ifTrue:ifFalse:
              {(set robot1free false) 1}
              {(set robot2free false) 2}))
    (method releaseRobot: (t)
         ((t = 1) ifTrue:ifFalse:
                  {(set robot1free true)}
                  {(set robot2free true)}))
)
```

The private `initLab` method ensures that in a new lab, both robots are available.

The class `RobotLabSimulation`

The class `RobotLabSimulation` maintains the state associated with a robot-lab simulation. A simulation carries a lot of internal state:

S144b. ⟨*simulation classes* S140⟩+≡ ◁ S144a S146c ▷

```
(class RobotLabSimulation
    [subclass-of Simulation]
    [ivars
     time-limit          ; time limit for using one robot
     lab                 ; current state of the lab
     robot-queue         ; the line of students waiting for a robot
     students-entered    ; the number of students who have entered the lab
     students-exited     ; the number of students who have finished and left
     timeWaiting         ; total time spent waiting in line by students
                         ; who have finished
     student-factory     ; class used to create a new student when one enters
     interarrival-times  ; stream of times between student entries
    ]
    ⟨methods of class RobotLabSimulation S145a⟩
)
```

Time limit t governs how long a student may use a robot while other students are waiting. But what happens in the lab is affected by more than just t. It also matters how many students there are, when students arrive at the lab, and how much time with a robot each student needs. All this information must be provided to the `RobotLabSimulation` object.

The number of students and the times at which they arrive are built into a single abstraction: a stream of *interarrival times*. (An interarrival time is the amount of time that elapses between the arrival of one student and the next.) The time needed by a student is built into a *factory* object that produces new students on demand. To create a simulation, then, I pass three parameters: a time limit t, a student factory `s`, and a stream of interarrival times `as`.

S145a. ⟨*methods of class* `RobotLabSimulation` S145a⟩≡ (S144b) S145b ▷

```
(class-method withLimit:student:arrivals: (t s as)
    ((super new) init-t:s:as: t s as))
(method init-t:s:as: (t s as) ; private method
    (set time-limit        t)
    (set student-factory   s)
    (set interarrival-times as)
    self)
```

The rest of the instance variables are initialized when the simulation is started by the `startUp` method. This method also initializes the superclass and schedules the (recurring) student arrivals.

S145b. ⟨*methods of class* `RobotLabSimulation` S145a⟩+≡ (S144b) ◁ S145a S145c ▷

```
(method startUp ()
    (set lab              (Lab new))
    (set students-entered 0)
    (set students-exited  0)
    (set timeWaiting      0)
    (set robot-queue      (Queue new))
    (super startUp)
    (self scheduleRecurringEvents:using: student-factory interarrival-times)
    self)
```

Finally, to prevent anybody from accidentally creating a simulation without initializing `time-limit`, `student-factory`, and `interarrival-times`, I redefine class method `new`:

S145c. ⟨*methods of class* `RobotLabSimulation` S145a⟩+≡ (S144b) ◁ S145b S145d ▷

```
(class-method new () (self error: 'robot-lab-simulation-needs-arguments))
```

My `finishUp` method reports on the results of the simulation. I print just the information I care about: the number of students who have finished, the number left in line, and the total and average times spent waiting by the students who finished.

S145d. ⟨*methods of class* `RobotLabSimulation` S145a⟩+≡ (S144b) ◁ S145c S146a ▷

```
(method finishUp ()
    ('Num-finished= print) (students-exited print)
    (self printcomma)
    ('left-waiting= print) ((robot-queue size) print)
    (self printcomma)
    ('total-time-waiting= print) (timeWaiting print)
    (self printcomma)
    ('average-wait= print) ((timeWaiting div: students-exited) println)
    (super finishUp))
(method printcomma () ; private
    (', print) (space print))
```

`div:`	*B* 651
`finishUp`	S141c
`ifTrue:ifFalse:`	*B* 639
`Object`	*B* 637
`print,`	
`@Collection`	*B* 637
`@CoordPair`	615
`@Student`	S147
`schedule-Recurring-Events:using:`	S143a
`Simulation`	S141a
`size`	*B* 644
`space`	*B* 641
`startUp`	S141b

At entry and exit, the simulation updates its internal statistics:

S146a. ⟨*methods of class* RobotLabSimulation S145a⟩+≡ (S144b) ◁S145d S146b▷

```
(method enter: (aStudent)
    (set students-entered (1 + students-entered)))
(method exit: (aStudent)
    (set students-exited  (1 + students-exited))
    (set timeWaiting      (timeWaiting + (aStudent timeWaiting))))
```

The `enter:` and `exit:` methods are called when events occur, not when they are scheduled. The `exit:` method relies on the `Student` object to be able to tell us how much time it has spent waiting in the queue.

The robot-lab simulation defines two resource methods: the `requestRobotFor:` method requests a robot for a student, and the `releaseRobot:` method gives it up.

S146b. ⟨*methods of class* RobotLabSimulation S145a⟩+≡ (S144b) ◁S146a S146d▷

```
(method requestRobotFor: (aStudent)
     ((lab hasARobot?) ifTrue:ifFalse:
          {(aStudent beGrantedRobot: (lab takeARobot))}
          {(robot-queue addLast: aStudent)}))

(method releaseRobot: (aRobot)
    (lab releaseRobot: aRobot)
    ((robot-queue isEmpty) ifFalse:
       {((robot-queue removeFirst) beGrantedRobot: (lab takeARobot))}))
```

These resource methods interact with a *queue*. If a student requests a robot when no robot is available, that student is put on the queue. And if, when a student releases a robot, there are other students waiting, the student who has been waiting the longest is removed from the queue and is granted use of the robot.

The robot queue is similar to the purely functional queue described in Section 2.6. But as is typical for Smalltalk, the queue is not a purely functional data structure; it is *mutable*. The operations I need from a queue (add at end and remove from beginning) are already provided by Smalltalk lists. But to help with debugging, I define a `Queue` subclass, which prints the list using the keyword `Queue`.

S146c. ⟨*simulation classes* S140⟩+≡ ◁S144b S147▷

```
(class Queue
    [subclass-of List]
)
```

Finally, the robot-simulation class exposes two public methods that make it possible for students to observe some of its state. The `time-limit` method makes it possible for a `Student` object to discover the time limit t, so it can relinquish its robot when the time limit expires. The `students-entered` method makes it easy to assign each `Student` object a unique number when it is created.

S146d. ⟨*methods of class* RobotLabSimulation S145a⟩+≡ (S144b) ◁S146b

```
(method time-limit       () time-limit)
(method students-entered () students-entered)
```

The class `Student`

In the robot-lab simulation, the active agents, also known as the simulation objects, are students. Each of these objects represents an individual who enters the lab, may wait in line, may use a robot, and so on. In the simulation, a student can be in one of four states: waiting for a robot, using robot 1, using robot 2, or finished. A diagram of these states, and of the messages that accompany transitions between them, is shown in Figure E.7.

`takeAction`	Simulate whatever action is appropriate to the receiver's current state.
`beGrantedRobot: aRobot`	Change the receiver's internal state to note that it now has a robot, and schedule a time at which to give up the robot.
`needsRobot?`	Answer whether the receiver still needs a robot.
`timeWaiting`	Answer the total amount of time the receiver has spent waiting for a robot.

(a) Instance protocol for `Student`

`timeNeeded`	This message is sent once, when an instance is created. The receiver answers the total amount of time it needs with a robot.
`relinquishRobot`	This method is sent when the receiver is using a robot, and time has arrived for the receiver to stop. In response, the receiver takes some action appropriate to its needs: if it is done with its work, it exits the simulation; otherwise it asks for more robot time.

(b) Private methods for `Student`

`new`	The class creates a new `Student` whose status is `'awaiting-robot`, and the `Student` immediately enters the active simulation and requests a robot from it.

(c) Class protocol for `Student`

Figure E.6: Protocol for `Student`

The `Student` class represents a student by six instance variables.

S147. ⟨*simulation classes* S140⟩+≡ ◁S146c S150c▷

```
(class Student
    [subclass-of Object]
    [ivars number         ; uniquely identifies this student
     status          ; 'awaiting-robot, 'finished, or a robot number
     timeNeeded      ; total work time this student needs
     timeStillNeeded ; time remaining for this student
     entryTime       ; time at which this student enters the simulation
     exitTime        ; time at which this student exits the simulation
    ]
    (method print () ('<Student print) (space print) (number print) ('> print))
    ⟨other methods of class Student S148⟩
)
```

These instance variables are used as follows:

- The `status` value indicates what the student is doing now, and also what it may do when it is next asked to do something via the `takeAction` method. The values are shown in Figure E.8, and they correspond to the oval states in Figure E.7.

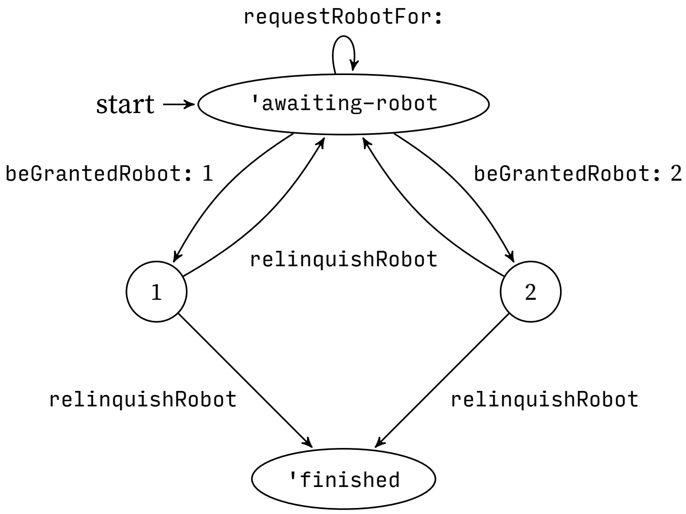


Figure E.7: State-transition diagram for a `Student`

Value	*State*
`'awaiting-robot`	Waiting for a robot (simulation will call `beGrantedRobot:`)
`1`	Using robot 1 (the next scheduled event is to release the robot)
`2`	Using robot 2 (the next scheduled event is to release the robot)
`'finished`	Finished (no more events will be scheduled for this student)

Figure E.8: Representation of the states in instance variable `status`

- Variable `timeNeeded` holds total amount of time the student needs with the robot in order to finish their lab work. Variable `timeStillNeeded` holds the amount of time left after whatever time the student has already spent with the robot. My simulation assumes that having the robot time broken into chunks doesn't affect the amount of time needed. In practice this assumption is probably false.

- Variables `entryTime` and `exitTime` provide an easy way to compute the total time the student spent in the lab. The difference between the total time and `timeNeeded` is the time spent waiting, which is the data I'm trying to gather. The data is provided to the simulation by the `timeWaiting` method.

  S148. ⟨*other methods of class* `Student` S148⟩≡ (S147) S149a ▷

  ```
  (method timeWaiting ()
      (exitTime - (entryTime + timeNeeded)))
  ```

To create a `Student` object, I use the classic pattern we have seen in classes `Picture` and `Shape`: a class method creates the instance, then executes a private method to initialize the object. Initialization is mostly straightforward: set the instance variables, enter the simulation, and ask for a robot. (Ideally, initialization would also be factored into small chunks that could be overridden separately.) But there's a little something extra going on with `timeNeeded` (next page).

S149a. ⟨*other methods of class* `Student` S148⟩+≡ (S147) ◁S148 S149b▷

```
(method timeNeeded () (self subclassResponsibility))
(class-method new () ((super new) init))
(method init () ; private
  (set number          (1 + (ActiveSimulation students-entered)))
  (set status          'awaiting-robot)
  (set timeNeeded      (self timeNeeded))
  (set timeStillNeeded timeNeeded)
  (set entryTime       (ActiveSimulation time-now))
  (ActiveSimulation enter: self)
  (ActiveSimulation requestRobotFor: self)
  self)
```

The value of instance variable `timeNeeded` is obtained by sending the `timeNeeded` *message* to `self`. What's going on here? My design uses different subclasses of `Student` to represent students who have different needs for the robot. By delegating the knowledge of the need to a subclass, I make it easy to run simulations with students who have different needs.

After it requests a robot, a `Student` cannot do anything until it is told. It waits to receive a `takeAction` message from the `RobotLabSimulation`, at which point its action depends on its `status`.

S149b. ⟨*other methods of class* `Student` S148⟩+≡ (S147) ◁S149a S149c▷

```
(method takeAction ()
   ((status = 'awaiting-robot) ifTrue:ifFalse:
      {(ActiveSimulation requestRobotFor: self)}
      {(self relinquishRobot)}))
```

A student who needs a robot asks for one. A student who doesn't need a robot must already have one. That student should give up the robot, by sending himself the `relinquishRobot` message.

Relinquishing a robot always returns the robot to the active simulation, by sending the `releaseRobot:` message. The rest of the action depends on the student's needs.

- If they need more time, they put themself in the `'awaiting-robot` state, and they immediately request the robot again. (They'll either wait in the queue, or in the special case where nobody else is waiting, they'll be granted the robot immediately. Because sending `requestRobotFor:` might result in an immediate message of `beGrantedRobot`, it's crucial that `status` be set to `'awaiting-robot` *before* `requestRobotFor:` is sent. Otherwise, the simulation might get into an inconsistent state in which the `Student` has been granted a robot but doesn't know it.)
- If the student has finished, they note the current time as the `exitTime` from the simulation, and then they exit the simulation. Again, order of evaluation is crucial: sending `exit:` will result in the simulation sending `timeWaiting`, and if `exitTime` has not been set, a run-time error will occur.

These choices are shown graphically in Figure E.7 by the two different arrows out of states 1 and 2, both labeled `relinquishRobot`.

S149c. ⟨*other methods of class* `Student` S148⟩+≡ (S147) ◁S149b S150a▷

```
(method relinquishRobot ()
     (ActiveSimulation releaseRobot: status)
     ((self needsRobot?) ifTrue:ifFalse:
          {(set status 'awaiting-robot)
           (ActiveSimulation requestRobotFor: self)}
          {(set status   'finished)
           (set exitTime  (ActiveSimulation time-now))
           (ActiveSimulation exit: self)}))
```

A student needs a robot if the time still needed is nonzero.

S150a. ⟨*other methods of class* `Student` S148⟩+≡ (S147) ◁S149c S150b▷

```
(method needsRobot? () (timeStillNeeded > 0))
```

The last remaining action in the `Student` class shows what happens when a student is granted use of a robot. They keep the robot for as long as needed, or for the time limit t, whichever is smaller. The `beGrantedRobot:` method saves this time interval in the local variable `time-to-use`. The `Student` object then adjusts its internal `timeStillNeeded`, changes its status, and schedules itself on the event queue. When the scheduled event arrives, the student's `takeAction` method will relinquish the robot.

S150b. ⟨*other methods of class* `Student` S148⟩+≡ (S147) ◁S150a

```
(method beGrantedRobot: (aRobot) [locals time-to-use]
     (set time-to-use (timeStillNeeded min: (ActiveSimulation time-limit)))
     (set timeStillNeeded (timeStillNeeded - time-to-use))
     (set status aRobot)
     (ActiveSimulation scheduleEvent:after: self time-to-use))
```

E.3.4 Running robot-lab simulations

To create a robot-lab simulation, I need a time limit, a student class, and a stream of interarrival times. I can then run the simulation for any given number of minutes. In a serious simulation, I would put a lot of effort into the classes that represent students' needs and arrival times. I would study how real students behave, create a probabilistic model, and code the model in Smalltalk. But studies are expensive, and force-feeding you a lot of probability and statistics would not help you learn about object-oriented techniques for implementing simulations. So I've chosen simplicity over realism; I make assumptions that oversimplify what happens in the real robot lab.

My first simplifying assumption is that every student needs two hours of robot time, which I measure in minutes:

S150c. ⟨*simulation classes* S140⟩+≡ ◁S147 S150d▷

```
(class Student120 [subclass-of Student]
           ; a student needing 120 minutes of robot time
    (method timeNeeded () 120)
)
```

My second simplifying assumption is that there are 20 students, and they all pour into the lab the moment it opens (i.e., when the simulation starts). I need to embody this assumption as an infinite stream of interarrival times. In other words, I need an object which, when it is sent the `next` message, will answer `0`. But only 20 times! After responding 20 times with `0`, the object should respond to future `next` messages with a very large time—one large enough to exceed the duration of any reasonable simulation. The object will be an instance of class `TwentyAtZero`:

S150d. ⟨*simulation classes* S140⟩+≡ ◁S150c S151b▷

```
(class TwentyAtZero [subclass-of Object] ; Twenty arrivals at time zero
    [ivars num-arrived]
    (class-method new () ((super new) init))
    (method init () (set num-arrived 0) self)
    (method next ()
         ((num-arrived = 20) ifTrue:ifFalse:
             {99999}
             {(set num-arrived (1 + num-arrived))
              0}))
)
```

I use these classes, plus my implementation of `PriorityQueue` from Exercise 1, to create a simulation `sim30`. I then run the simulation for 20 simulated hours:

S151a. ⟨*simulation transcript* S151a⟩≡ S152▷

```
-> (use pqueue.smt) ; implementation of class PriorityQueue
-> (use sim.smt)    ; implementations of the simulation classes
-> (val sim30 (RobotLabSimulation withLimit:student:arrivals: 30 Student120
                                              (TwentyAtZero new)))
-> (sim30 runUntil: 1200)
Num-finished=20, left-waiting=0, total-time-waiting=18900, average-wait=945
<RobotLabSimulation>
```

The robot lab was open long enough to serve all 20 students, and they all finished. But the 30-minute time limit lead to long waits: the average student waits for 945 minutes, spending nearly eight times as much time in line as working with a robot. The results of all four runs are as follows:

Time limit t	Students served	Students left waiting	Average wait time
30	20	0	945
60	20	0	810
90	20	0	945
120	20	0	540

If I want to minimize average waiting time, I do best to let each student monopolize a robot for a full two hours. This policy may not be fair, but it's efficient.

What if not all students are alike? Let's assume that only half the students need two hours each. The other half are accomplished roboticists and can finish their work in half an hour. Every time I create a new `Student`, I'll assume that the time needed by the new `Student` is 150 minutes minus the time needed by the previous student. That works out to `Students` who alternate between needing 120 minutes and 30 minutes.

S151b. ⟨*simulation classes* S140⟩+≡ ◁S150d S151c▷

```
(val last-student-needed 30) ; time needed by last created AlternatingStudent
(class AlternatingStudent
    [subclass-of Student]
    (method timeNeeded ()
         (set last-student-needed (150 - last-student-needed))
         last-student-needed)
)
```

In Smalltalk-80 I would store `last-student-needed` in a *class variable*, which would be shared among all instances of `AlternatingStudent`.

Let's also assume that the students know that there are only two robots, so they don't all crowd into the lab when it opens. Instead, they arrive every 35 minutes. And to keep the implementation simple, I won't cap the number of students at 20; instead, I assume that as long as the lab is open, students keep coming.

An object of class `EveryNMinutes` always returns the same interarrival time `n`, which is passed as a parameter to class method `new:`.

S151c. ⟨*simulation classes* S140⟩+≡ ◁S151b

```
(class EveryNMinutes
    [subclass-of Object]
    [ivars interval]
    (class-method new: (n) ((super new) init: n))
    (method init: (n) (set interval n) self)
    (method next () interval)
)
```

To make these new simulations easier to run, I create an auxiliary helper class `AlternatingLabSim`. It's a subclass of `RobotLabSimulation`, and it has an extra class method which knows to use `AlternatingStudent` every 35 minutes. Again, I run it four times:

S152. ⟨*simulation transcript* S151a⟩+≡ ◁S151a

```
-> (class AlternatingLabSim
      [subclass-of RobotLabSimulation]
      (class-method runWithLimit: (n)
         ((super withLimit:student:arrivals:
                 n
                 AlternatingStudent
                 (EveryNMinutes new: 35))
            runUntil: 1200))
   )
-> (AlternatingLabSim runWithLimit: 30)
Num-finished=30, left-waiting=2, total-time-waiting=1095, average-wait=36
<AlternatingLabSim>
-> (AlternatingLabSim runWithLimit: 60)
Num-finished=30, left-waiting=2, total-time-waiting=1235, average-wait=41
<AlternatingLabSim>
-> (AlternatingLabSim runWithLimit: 90)
Num-finished=29, left-waiting=3, total-time-waiting=1190, average-wait=41
<AlternatingLabSim>
-> (AlternatingLabSim runWithLimit: 120)
Num-finished=30, left-waiting=2, total-time-waiting=1120, average-wait=37
<AlternatingLabSim>
```

The new results are:

Time limit t	Students served	Students left waiting	Average wait time
30	30	2	36
60	30	2	41
90	29	3	41
120	30	2	37

The glacial wait times have been eliminated, and with these different students, there's no time limit t that is clearly superior. Both the 30-minute "rapid turnover" and 120-minute "hold for two hours" policies appear about 12% better than other limits, but because the simulation is so unrealistic, I shouldn't draw any conclusions.

E.3.5 Summary and analysis

My simulation omits too many details. For example, a real student who enters the lab and finds a long line may *balk*, i.e., they may leave and try again later. I don't consider the cost of interruptions; a student whose work is broken into several sessions may need more time with the robots.[3] "Average time waiting" is not a definitive measure for comparing time limits, because it values everyone's time equally. But Professor S might prefer a policy under which students who need less time don't have to wait as long as students who need more time.

Most importantly, my simulations make bogus assumptions about needs and about arrival times—and these assumptions probably have a decisive effect on the

[3] It's also possible that students who are interrupted spend more time thinking, after which they may need to spend *less* time fiddling with robots.

results. I might build into the simulation a list of needs and arrival times obtained by observing real students, or I might simply invent a probabilistic model that I believe better reflects the needs of real students, then generate students randomly from the model.

Many of the problems enumerated above can be addressed by making modest changes to the simulation code. Suggestions for such changes appear in Exercise 3.

Although my simulation does not accurately model real students working in real labs, it *does* demonstrate a good way to organize an object-oriented simulation. To understand the organization deeply, you will need to do some exercises. But I can jump-start your understanding by looking at the organization through the lens of a single computation: the algorithm executed when a new student enters the lab. In a typical procedural language like C or Impcore, I might write a single "new student" procedure that does this:

- Allocate memory for the student and initialize its fields. Increment the number of students in the simulation. Finally check to see if a robot is available. If a robot is available, assign it to the student and add a "robot time expires" event to the event queue. If no robot is available, put the student on the queue for the robot.

Let's contrast this single "new student" procedure with the way the same computation is done in the Smalltalk code:

1. An object of class `RecurringEvents` sends a `new` message to its local factory, which is the class object `Student120`.

2. The `new` message is dispatched to class `Student`, which sends `(super new)`, which is dispatched to `Object`. Space is allocated for the object and its instance variables. The `new` method in class `Student` then sends `init` to the new object.

3. The `init` method on class `Student` initializes the instance variables, which includes sending `timeNeeded` to `self`, which dispatches on the `Student120` class, answering `120`. The init method then sends `enter:` to the active simulation.

4. The `enter:` method on class `RobotLabSimulation` increments the number of students in the simulation.

5. The `init` method on class `Student` finishes by sending `requestRobotFor:` to the active simulation.

6. The `requestRobotFor:` method on class `RobotLabSimulation` checks to see if a robot is available. If a robot is available, it notes that the robot is no longer free, then sends `beGrantedRobot:` to the student; otherwise it adds the student to the robot queue.

7. The `beGrantedRobot:` method on class `Student` notes that the student is using the robot, calculates a `time-to-use`, then sends `scheduleEvent:after:` to the active simulation.

8. The `scheduleEvent:after:` method dispatches to the superclass `Simulation`, which in turn dispatches to `scheduleEvent:at:`, which finally puts the "robot time expires" event on the event queue.

`Alternating-Student`	S151b
`EveryNMinutes`	S151c
`RobotLab-Simulation`	S144b
`runUntil:`	S142a

This example illustrates what's hard about object-oriented programming: the algorithm, which the procedural programmer thinks of as one simple sequence of actions, ends up being "smeared out" over nine methods defined on four classes. But because the pieces of the algorithm are distributed over four classes, it is much easier to reuse the pieces—and it is easy, via inheritance, to create variants of the classes, such as students with different behaviors. Learning to create this sort of design is the key to becoming a productive object-oriented programmer.

E.3.6 Robot-lab exercises

Exercises 1 and 2 ask you to implement the priority queue needed to run simulations. Exercises 3 to 7 invite you to explore discrete-event simulation in more depth. Exercise 3 suggests a number of ways to make the robot-lab simulation (Section E.3) more realistic. Exercise 4 asks you to improve the resource-handling code so that it can be written once and used for many simulations. Exercise 5 asks you to develop better ways of generating streams of events. Exercise 6 asks you to create new `Student` objects using a *factory object* rather than a class. Finally, Exercise 7 asks you to repair a defect in the design of the `Simulation` class.

1. The discrete-event simulation requires a priority queue, whose protocol is given in Figure E.5 on page S141. Use the variable-size arrays from Exercise 23 in Chapter 10 (page 720) to implement class `PriorityQueue`:

   (a) As your representation, use a variable-size array that holds a sequence of `Associations`. In each `Association`, the value represents an event, and the key represents the time at which the event is scheduled to occur.

   (b) Maintain the invariant that the array is sorted by event time. You can then implement `removeMin` using `remlo`, and you can implement `at:put:` by using `addhi:` and then sifting down the new element into its new position in the array.

   (c) Prove that this implementation takes constant time for `removeMin` and $O(n)$ time for `at:put:`, where n is the number of elements in the queue.

2. If I'm implementing a priority queue, I can do better than $O(n)$ time for insertion. You can also do better if you store the queue's elements in an array which is indexed from 1 to n and which satisfies the following invariant:

   $$\forall k. a[k] \le a[2k] \land a[k] \le a[2k+1],$$

   whenever $2k \le n$ and $2k+1 \le n$. This is the same invariant that is used in the detailed priority-queue example in Chapter 9 (Section 9.6.5, page 551).

   (a) Prove that the invariant implies that $a[1]$ is the smallest element of the array.

   (b) Prove that removing the last element maintains the invariant.

   (c) If the first element is replaced by an arbitrary element, the invariant can be re-established by the following procedure:

   ```
   let $k = 1$
   while ($2k \le n$ and $a[k] > a[2k]$) or ($2k + 1 \le n$ and $a[k] >$
   $a[2k + 1]$) do
     swap $a[k]$ with the smaller of $a[2k]$ and $a[2k + 1]$
     replace $k$ with $2k$ or $2k + 1$, whichever was used to swap
   ```

If an arbitrary element is added at the end, the invariant can be established by similar procedure involving repeated swapping with $a[\lfloor \frac{k}{2} \rfloor]$.

(d) Use these facts to implement a priority queue. You can use the extensible arrays from Exercise 23, or you can implement a simpler extensible array that grows and shrinks only at the right-hand side.

(e) Measure the effect on simulation time.

3. You could improve the robot-lab simulation in a number of ways:

(a) Professor S gets a big grant and buys three new robots, increasing the number in the lab to 5. Reimplement the `Lab` class so it can easily represent a lab containing 5 robots. Make sure that when robots wear out or future robots are acquired, the code will be easy to update. (Hint: μSmalltalk's initial basis includes class `Set`.)

(b) Define a new simulation `VerboseRobotLabSimulation`, which prints a message when a student leaves the lab. The message should identify the student, the time of arrival, and the time of departure. Don't touch any existing code. Remember `super`.

(c) Modify the model to allow for *balking*: assume that if a student arrives and finds more than five other students in line, the student leaves immediately. And account for time lost to interruptions: if a student has to relinquish a robot before having finished, that student now needs fifteen more minutes.

(d) When a student finishes, compute their *time-waiting ratio*: total time spent in the lab divided by time spent using robots. (To represent the ratio, use `Fraction` or `Float`.) At the end of a simulation, report on the largest time-waiting ratio suffered during that simulation. As a measure of quality, how does time-waiting ratio compare with average waiting time? Do they agree on the best policy?

Solve this problem without modifying existing code—just define new subclasses.

(e) Student arrivals should be random. A process of random arrivals occurring at a fixed rate is called a *Poisson process*. In a Poisson process, the probability density function for interarrival times Δt is an exponential $e^{-\lambda\Delta t}$, where λ is the arrival rate measured in students per minute. If you have a way of generating random floating-point numbers U over the unit interval $[0.0, 1.0]$, you can compute a suitably distributed Δt by using the equation

$$\Delta t = \frac{-\ln U}{\lambda}.$$

Implement a `PoissonEveryNMinutes` class which uses random numbers to deliver *random* interarrival times with an expected rate of $\frac{1}{N}$ students per minute. To compute the natural logarithm in μSmalltalk you can either use an approximation method suited to computing the log of a number between 0 and 1, or you can modify the interpreter to add a primitive logarithm based on the Standard ML function `Math.ln`, which operates on floating-point numbers.

4. In the discrete-event simulation, robots are *fungible*. That is, one robot is as good as any other robot, and as long as a `Student` object gets a robot, it doesn't matter which one. Simulations turn out to be full of fungible resources: luggage carts, Boeing 747s, gallons of gasoline, twenty-dollar bills, and more. There is no reason that every new simulation class should have to implement code to manage fungible resources—it should be done once in a superclass.

   Design and implement methods on class `Simulation` that allow simulation objects to manage arbitrary collections of named, fungible resources. You might consider some of the following methods:

   - A method that requests a single resource (or N units of resource) by name.
   - A method that returns resources.
   - A method that makes a resource name *known* to the simulation. Attempts to request or return resources with unknown names should cause run-time errors.
   - Methods that tell the simulation to create or destroy resources.

   In addition, you will have to expand the protocol for simulation objects so that any simulation object can be granted resources by name.

   Your implementation should generalize the code in the robot-lab simulation: if a simulation object requests an available resource, the request should be granted right away; if a simulation object requests an unavailable resource, the object should be put onto a queue associated with the resource.

   To check your work, you can reimplement the robot-lab simulation using your new methods.

5. In the discrete-event simulation, the implementation of streams should offend you: there is no composition and no reuse. Design and implement a library of stream classes that offer the following functionality:

   (a) Implement a superclass `Stream` that includes the collection methods `select:`, `reject:`, and `collect:`. Method `next` should be a subclass responsibility.

   (b) Implement a subclass stream s in which something occurs every n minutes. That is, sending `next` always answers n.

   (c) Given a stream s and a limit N, produce a new stream s' that such that repeatedly sending `next` produces the first N elements of s and afterward answers only `nil`.

   (d) Given two streams s_1 and s_2, produce a new stream s such that repeatedly sending `next` to s produces first all the elements of s_1, followed by all the elements of s_2.

   (e) Given two streams s_1 and s_2, produce a new stream s such that repeatedly sending `next` to s produces alternating elements of s_1 and s_2 (that is, s_1 and s_2 "take turns").

   (f) Use your library to reimplement the streams used in the discrete-event simulation.

6. In the discrete-event simulation, when I have a new model of students' needs, I have to create a new subclass of class `Student`. Creating these classes is tedious, and this coding style makes it unnecessarily hard to, for example,

read needs from a file. Address these problems by creating a single class `StudentFactory`, such that

- To create `StudentFactory`, you supply a stream of needs to a class method `new:`.
- An *instance* of class `StudentFactor` can respond to a `new` message, which it does by pulling the "time needed" from its stream, then creating and answering a new instance of `Student` with that need.

Try creating a subclass of `Student` that works with the `StudentFactory`.

The idea of using an object to create other objects is so popular that "Factory" is used as the name of a *design pattern*.

7. The `Simulation` class in Section E.3.2 is not well designed: although the `startUp`, `proceed`, and `finishUp` methods provide a handy way to organize initialization and finalization, they can't actually be used by clients, because if the event queue happens to be empty, it's not safe to call `proceed`. Repair this defect by changing class `Simulation`. Change the implementation, and if necessary, change the protocol as well.

Part V. Interesting infrastructure

Appendix F contents

Code for writing interpreters in C F

Lots of people are interested in this book because they are interested in interpreters. But the implementation of interpreters isn't the main event; code in Chapters 1 and 2 and elsewhere shows only the parts of the Impcore and Scheme interpreters that are most relevant to the study of programming languages. That code is the tip of an iceberg, and there's a good deal beneath the surface. Much of it is interesting, some is not. The parts that are generic to writing interpreters, not specific to Impcore, can be found here and in Appendix G.

This appendix presents most of the implementations of the interfaces shown in Chapter 1. It also presents interfaces and implementations used to read lines and parenthesized phrases from input. Everything presented here is used not only to help implement Impcore, but also to help implement μScheme and μScheme+ in Chapters 2 to 4. And almost everything used to implement Impcore is presented here—with two exceptions.

- The parsing code used to convert input to abstract syntax uses a form of *shift-reduce* parsing. While the technology is old and is well understood, it is still the most complicated part of the interpreter. The complexity is justified because it makes it easy for you to extend any of the parsers, but because the code is complex, it is best presented on its own. The parsing infrastructure is shown in Appendix G, along with its application to the Impcore parser.

- There are a few parts of the Impcore interpreter, like the functions that print abstract syntax, or the implementation of function environments, which are not reused in any other interpreter. These parts are relegated to Appendix K.

All the infrastructure presented in this appendix is reusable. If you use it to build your own interpreters, your interpreters will be simple and easy to modify, but not fast.

The code in this appendix is organized to parallel the presentation in Chapter 1. A detailed overview, which connects concepts, types, functions, interfaces, and implementations, is shown in Table F.1 (page S162). A higher-level overview, which shows what information is presented in each chapter or appendix, is shown in Table F.2 (page S163).

F.1 Names

The `Name` is an abstraction built from a string—a sequence of characters. Two names built from equal strings are *identical*, even if the strings were located in different parts of memory. That means that two names can be compared for equality using a constant-time pointer comparison, which is not possible with strings.

Table F.1: Key ideas, their interfaces, and their implementations (excludes parsing)

Abstract syntax, names, values, functions, and environments

Concept	Types & Functions	Interface	Implementation
Abstract syntax	`Exp`, `Def`	§1.6.1 (page 42)	(exposed rep)
Abstract syntax	`XDef`, `UnitTest`	§K.1.2 (page S292)	(exposed rep)
Names	`Name`	§1.6.1 (page 43)	§F.1 (page S164)
Value	`Value`	§1.6.1 (page 43)	(exposed rep)
Function	`Func`, `Userfun`	§1.6.1 (pages 42 & 44)	(exposed rep)
Environment	`Valenv`	§1.6.1 (page 44)	§1.6.3 (page 54)
Environment	`Funenv`	§1.6.1 (page 44)	§K.3.1 (page S303)

Evaluation

Concept	Types & Functions	Interface	Implementation
Evaluator	`eval`	§1.6.1 (page 45)	§1.6.2 (page 48)
Evaluator	`evaldef`	§1.6.1 (page 45)	§1.6.2 (page 53)
Evaluator	`readevalprint`	§K.1.4 (page S293)	§K.2.1 (page S296)
Interaction	`Echo`	§K.1.4 (page S293)	(exposed rep)

Streams and lists

Concept	Types & Functions	Interface	Implementation
Extended definitions	`XDefstream`, `filexdefs`, `stringxdefs`, `getxdef` `xdefstream`	§§F.2.3 and K.1.4 (pages S173 & S293)	§F.2.3 (pages S173 & S173)
Parenthesized phrases	`Par`, `Parstream`, `getpar`	§F.2.2 (page S168)	§F.2.2 (pages S169 & S170)
Lines	`Linestream`, `getline_`	§F.2.1 (page S165)	§F.2.1 (pages S166 & S167)
Lists of `Exp`s, `Value`s, and others	(not shown)	§1.6.1 (page 45)	(generated automatically)

Printing and error signaling

Concept	Types & Functions	Interface	Implementation
Printers	`print`, `fprint`	§1.6.1 (page 46)	§F.4.1 (page S178)
Error-signaling printers	`synerror`, `runerror`, `othererror`	§§1.6.1 & 1.6.1 (pages 47 & 47)	§F.5.1 (pages S182 & S183)
Error helpers	`checkargc`, `duplicatename`	§§1.6.1 & F.5.2 (pages 47 & S184)	§F.5.2 (page S184)
Printer extension	`installprinter`, `Printer`	§§F.4 and K.1.5 (pages S177 & S293)	§F.4.2 (page S179)
Source locations	`Sourceloc`	§K.1.6 (page S293)	(exposed rep)
Error formats	`ErrorFormat`	§K.1.6 (page S294)	(exposed rep)
Error modes	`ErrorMode`, `set_error_mode`	§F.5 (page S181)	§F.5.1 (page S181)

Table F.2: The implementation of Impcore (chapters, appendices, and files)

Chapter 1: central ideas and fundamental data structures

Lines	Where	What
	`all.h`	Representations of `Exp`, `Def`, `XDef`, `Value`, and lists
53	`env.c`	Operations on value environments
369	`eval.c`	Evaluation: `eval`, `evaldef`, `readevalprint`
68	`impcore.c`	The `main` function (launches the interpreter)
45	`name.c`	Conversion between names and strings, used in many interpreters

Appendix F: (mostly) reusable code for writing interpreters in C

Lines	Where	What
92	`error.c`	Error functions, formats, modes
176	`lex.c`	Get `Par` from string, `Linestream` using `getpar`, `getparlist`
18	`overflow.c`	Detect stack overflow
67	`print.c`	The extensible printer
86	`linestream.c`	Build `Linestream`s from files or strings; `getline_`
31	`tests.c`	Report test results
33	`xdefstream.c`	Functions `xdefstream` and `getxdef`

Appendix G: code for parsing, both reusable and specific to Impcore

Lines	Where	What
111	`parse.c`	Impcore-specific code and parsing tables, turn `Par` into `Exp` or `XDef`
347	`tableparsing.c`	Reusable infrastructure: `tableparse`, `rowparse`, common shift functions

Appendix K: code that is peripheral to the ideas and is specific to Impcore

Lines	Where	What
50	`env.c`	Operations on function environments
103	`printfuns.c`	Printing functions for `Value`, `Exp`, `XDef`, many others
67	`imptests.c`	Run unit tests using Impcore's dual environments

Each name is associated with a string it was built from. The string is stored inside the name.

S164a. ⟨*name.c* S164a⟩≡ S164b ▷

```
struct Name {
    const char *s;
};
```

Returning the string associated with a name is trivial.

S164b. ⟨*name.c* S164a⟩+≡ ◁ S164a S164c ▷

```
const char* nametostr(Name np) {
    assert(np != NULL);
    return np->s;
}
```

Finding the name associated with a string is harder. To meet the specification, if `strtoname` gets a string it has seen before, it must return the same name it returned before. To remember what it has seen and returned, it uses the simplest possible data structure: `all_names`, a list of all the names ever returned. Given a string `s`, a simple linear search finds the name associated with it, if any.

S164c. ⟨*name.c* S164a⟩+≡ ◁ S164b

```
Name strtoname(const char *s) {
    static Namelist all_names;
    assert(s != NULL);

    for (Namelist unsearched = all_names; unsearched; unsearched = unsearched->tl)
        if (strcmp(s, unsearched->hd->s) == 0)
            return unsearched->hd;

    ⟨allocate a new name, add it to all_names, and return it S164d⟩
}
```

A faster implementation might use a search tree or a hash table, not a simple list. Such an implementation is described by Hanson (1996, chapter 3).

If the string `s` isn't associated with any name on the list `all_names`, then `strtoname` makes a new name and adds it.

S164d. ⟨*allocate a new name, add it to* `all_names`*, and return it* S164d⟩≡ (S164c)

```
Name np = malloc(sizeof(*np));
assert(np != NULL);
np->s = malloc(strlen(s) + 1);
assert(np->s != NULL);
strcpy((char*)np->s, s);
all_names = mkNL(np, all_names);
return np;
```

F.2 Streams

An evaluator works by repeatedly calling `getxdef` on a stream of `XDef`s. Behind the scenes, there's a lot going on:

- Each `XDef` is produced from a parenthesized phrase, for example something like `(val n 0)` or `(define id (x) x)`. A parenthesized phrase, which in the code is called `Par`, is simply a fragment of the input in which parentheses are balanced; converting a parenthesized phrase to an expression or an extended definition is the job of the parser presented in Appendix G. Producing parenthesized phrases, however, is done here; function `parstream` produces a stream of `Par`s, called `Parstream`, and `getpar` takes a `Parstream` and produces a `Par`.

- A Par is found on one or more input lines. (And an input line may contain more than one Par.) A Parstream is produced from a Linestream, and a Linestream may be produced either from a string or from an input file.

Each stream follows the same pattern: there are one or more functions to create streams, and there's a function to get a thing from a stream. Their implementations are also similar. All the streams and their implementations are presented in this section. I present streams of lines first, then parenthesized phrases, and finally extended definitions. That way, as you read each implementation, you'll be familiar with what it depends on.

F.2.1 Streams of lines

A Linestream encapsulates a seqeuence of input lines.

Interface to Linestream

To use a Linestream, call getline_.[1] The getline_ function prints a prompt, reads the next line of input from the source, and returns a pointer to the line. Client code needn't worry about how long the line is; getline_ allocates enough memory to hold it. Because getline_ reuses the same memory to hold successive lines, it is an unchecked run-time error to retain a pointer returned by getline_ after a subsequent call to getline_. A client that needs to save input characters must copy the result of getline_ before calling getline_ again.

S165a. ⟨*shared type definitions* S165a⟩≡ (S295a) S168c ▷

```
typedef struct Linestream *Linestream;
```

S165b. ⟨*shared function prototypes* S165b⟩≡ (S295a) S165c ▷

```
char *getline_(Linestream r, const char *prompt);
```

A Linestream can be created from a string or a file. And when a Linestream is created, it is given a name for the string or file; that name is used in error messages.

S165c. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁ S165b S168e ▷

```
Linestream stringlines(const char *stringname, const char *s);
Linestream filelines  (const char *filename,   FILE *fin);
```

If an s passed to stringlines is nonempty, it is a checked run-time error for it to end in any character except newline. After a call to stringlines, client code must ensure that pointers into s remain valid until the last call to getline_. If getline_ is called after the memory pointed to by s is no longer valid, it is an *unchecked* run-time error.

mkNL	$\mathcal{A}$
type Name	43a
type Namelist	43a

[1] The function is called getline_ with a trailing underscore so as not to conflict with getline, a POSIX standard function. I was using getline for 20 years before the POSIX function was standardized, and I'm too stubborn to change.

Implementation of `Linestream`

A `Linestream` owns the memory used to store each line. That memory is pointed to by `buf`, and its size is stored in `bufsize`. If no line has been read, `buf` is `NULL` and `bufsize` is zero.

S166a. ⟨*shared structure definitions* S166a⟩≡ (S295a)

```
struct Linestream {
    char *buf;               // holds the last line read
    int bufsize;             // size of buf

    struct Sourceloc source; // where the last line came from
    FILE *fin;               // non-NULL if filelines
    const char *s;           // non-NULL if stringlines
};
```

The rest of the `Linestream` structure stores mutable state characterizing the source from which lines come:

- The `source` field tracks the location of the line currently in `buf`.
- The `fin` field, if the stream is built from a file, contains the pointer to that file's handle. Otherwise `fin` is `NULL`.
- The `s` field, if the stream is built from a string, points to the characters of that string that have not yet been converted to lines. Otherwise `s` is `NULL`.

The stream-creator functions do the minimum needed to establish the invariants of a `Linestream`. To clear fields that should be zero, they use the standard C function `calloc`.

S166b. ⟨*linestream.c* S166b⟩≡ S166c ▷

```
Linestream stringlines(const char *stringname, const char *s) {
    Linestream lines = calloc(1, sizeof(*lines));
    assert(lines);
    lines->source.sourcename = stringname;
    ⟨check to see that s is empty or ends in a newline S166d⟩
    lines->s = s;
    return lines;
}
```

S166c. ⟨*linestream.c* S166b⟩+≡ ◁ S166b S167a ▷

```
Linestream filelines(const char *filename, FILE *fin) {
    Linestream lines = calloc(1, sizeof(*lines));
    assert(lines);
    lines->source.sourcename = filename;
    lines->fin = fin;
    return lines;
}
```

S166d. ⟨*check to see that* `s` *is empty or ends in a newline* S166d⟩≡ (S166b)

```
{   int n = strlen(s);
    assert(n == 0 || s[n-1] == '\n');
}
```

Function `getline_` returns a pointer to the next line from the input, which is held in `buf`, a buffer that is reused on subsequent calls. Function `growbuf` makes sure the buffer is at least `n` bytes long.

S167a. ⟨*linestream.c* S166b⟩+≡ ◁S166c S167b▷

```
static void growbuf(Linestream lines, int n) {
    assert(lines);
    if (lines->bufsize < n) {
        lines->buf = realloc(lines->buf, n);
        assert(lines->buf != NULL);
        lines->bufsize = n;
    }
}
```

Here's a secret: I've tweaked `getline_` to check and see if the line read begins with the special string `;#`. If so, the line is printed. This string is a special comment that helps me test all the ⟨*transcript*⟩ examples in the book.

S167b. ⟨*linestream.c* S166b⟩+≡ ◁S167a

```
char* getline_(Linestream lines, const char *prompt) {
    assert(lines);
    if (prompt)
        print("%s", prompt);

    lines->source.line++;
    if (lines->fin)
        ⟨set lines->buf to next line from file lines->fin, or return NULL if lines are exhausted S167c⟩
    else if (lines->s)
        ⟨set lines->buf to next line from string lines->s, or return NULL if lines are exhausted S168a⟩
    else
        assert(0);

    if (lines->buf[0] == ';' && lines->buf[1] == '#')
        print("%s\n", lines->buf);

    return lines->buf;
}
```

To get a line from a file, `getline_` calls the C standard library function `fgets`. If the buffer is big enough, `fgets` returns exactly the next line. If the buffer isn't big enough, `getline_` grows the buffer and calls `fgets` again, to get more of the line. This process is iterated until the last character in the buffer is a newline. Then `getline_` chops off the terminating newline by overwriting it with `'\0'`.

S167c. ⟨*set* `lines->buf` *to next line from file* `lines->fin`*, or return* `NULL` *if lines are exhausted* S167c⟩≡ (S167b)

```
{
    int n; /* number of characters read into the buffer */

    for (n = 0; n == 0 || lines->buf[n-1] != '\n'; n = strlen(lines->buf)) {
        growbuf(lines, n+512);
        if (fgets(lines->buf+n, 512, lines->fin) == NULL)
            break;
    }
    if (n == 0)
        return NULL;
    if (lines->buf[n-1] == '\n')
        lines->buf[n-1] = '\0';
}
```

type Linestream	S165a
print	S176d

When reading from a string, `getline_` looks in `lines->s`. It finds the next newline, copies the intervening characters into `buf`, and updates `lines->s`.

S168a. ⟨*set* `lines->buf` *to next line from string* `lines->s`*, or return* `NULL` *if lines are exhausted* S168a⟩≡ (S167b)

```
{
    const char *p = strchr(lines->s, '\n');
    if (p == NULL)
        return NULL;
    p++;
    int len = p - lines->s;
    growbuf(lines, len);
    strncpy(lines->buf, lines->s, len);
    lines->buf[len-1] = '\0';   /* no newline */
    lines->s = p;
}
```

F.2.2 Streams of parenthesized phrases

Calling a `Par` a "parenthesized phrase" doesn't tell the whole truth: the `Par` type includes not only phrases with balanced parentheses but also single atoms like `3`, `#t`, and `gcd`. In truth, a parenthesized phrase is one of the following:

- A single atom
- A list of zero or more parenthesized phrases, wrapped in parentheses.

In other words, it's a lot like an S-expression (Chapter 2).

A `Par` is defined as follows, using ASDL (Appendix J):

S168b. ⟨*par.t* S168b⟩≡

```
Par* = ATOM (Name)
     | LIST (Parlist)
```

S168c. ⟨*shared type definitions* S165a⟩+≡ (S295a) ◁S165a S168d▷

```
typedef struct Parlist *Parlist; /* list of Par */
```

This simple structure reflects the *concrete syntax* of Impcore, μScheme, and the other bridge languages. It's simple because I've stolen the simple concrete syntax that John McCarthy developed for Lisp. Simple syntax is represented by a simple data structure.

Interface to `Parstream`

A `Parstream` is an abstract type.

S168d. ⟨*shared type definitions* S165a⟩+≡ (S295a) ◁S168c S174a▷

```
typedef struct Parstream *Parstream;
```

To create a `Parstream`, client code specifies not only the lines from which `Par`s will be read but also the prompts to be used (page S293). To get a `Par` from a stream, client code calls `getpar`. And for error messages, client code can ask a `Parstream` for its current source location.

S168e. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S165c S169a▷

```
Parstream parstream(Linestream lines, Prompts prompts);
Par       getpar   (Parstream r);
Sourceloc parsource(Parstream pars);
```

The Parstream interface is completed by global variable read_tick_as_quote. When read_tick_as_quote is true, getpar turns an input like '(1 2 3) into the parenthesized phrase (quote (1 2 3)). When set, this variable makes the tick mark behave the way μScheme wants it to behave.

S169a. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S168e S174b▷

```
extern bool read_tick_as_quote;
```

In Impcore, a tick mark is not read as (quote ...), so read_tick_as_quote is false.

S169b. ⟨*impcore.c* S169b⟩≡

```
bool read_tick_as_quote = false;
```

Implementation of Parstream

The representation of a Parstream has three parts:

- The lines field is a source of input lines.
- The input field contains characters from an input line; if a Par has already been read from that line, input contains only the characters left over.
- The prompts structure contains strings that are printed every time a line is taken from lines. When the Parstream is reading a fresh Par, it issues prompts.ps1 for the first line of that Par. When it has to read a Par that spans more than one line, like a long function definition, it issues prompts.ps2 for all the rest of the lines. The names ps1 and ps2 stand for "prompt string" 1 and 2; they come from the Unix shell.

S169c. ⟨*lex.c* S169c⟩≡ S169d▷

```
struct Parstream {
    Linestream lines;     /* source of more lines */
    const char *input;    /* what's not yet read from the most recent input line */
    /* invariant: unread is NULL only if lines is empty */

    struct {
       const char *ps1, *ps2;
    } prompts;
};
```

Upon creation, a Parstream is initialized from the parameters to parstream. Initializing input to an empty string puts the stream into a state with no characters left over.

S169d. ⟨*lex.c* S169c⟩+≡ ◁S169c S169e▷

```
Parstream parstream(Linestream lines, Prompts prompts) {
    Parstream pars = malloc(sizeof(*pars));
    assert(pars);
    pars->lines = lines;
    pars->input = "";
    pars->prompts.ps1 = prompts == PROMPTING ? "-> " : "";
    pars->prompts.ps2 = prompts == PROMPTING ? "   " : "";
    return pars;
}
```

Function parsource grabs the current source location out of the Linestream.

S169e. ⟨*lex.c* S169c⟩+≡ ◁S169d S170▷

```
Sourceloc parsource(Parstream pars) {
    return &pars->lines->source;
}
```

Function getpar presents a minor problem: the Par type is defined recursively, so getpar itself must be recursive. But the first call to getpar is distinct from the others in two ways:

- If the first call prompts, it should use prompts.ps1. Other calls should use prompts.ps2
- If the first call encounters a right parenthesis, then the right parenthesis is unbalanced, and getpar should report it as a syntax error. If another call encounters a right parenthesis, then the right parenthesis marks the end of a LIST, and getpar should scan past it and return.

This distinction is managed by function getpar_in_context, which knows whether it is the first call or another call. This function attempts to read a Par. If it runs out of input, it returns NULL. If it sees a right parenthesis, it returns NULL if and only if is_first is false; otherwise, it calls synerror.

S170. ⟨*lex.c* S169c⟩+≡ ◁S169e S171a▷

```
⟨prototypes of private functions that help with getpar S171e⟩
static Par getpar_in_context(Parstream pars, bool is_first, char left) {
    if (pars->input == NULL)
        return NULL;
    else {
        char right;      // will hold right bracket, if any
        ⟨advance pars->input past whitespace characters S171b⟩
        switch (*pars->input) {
        case '\0':  /* on end of line, get another line and continue */
        case ';':
            pars->input = getline_(pars->lines,
                                   is_first ? pars->prompts.ps1 : pars->prompts.ps2);
            return getpar_in_context(pars, is_first, left);
        case '(': case '[':
            ⟨read and return a parenthesized LIST S171f⟩
        case ')': case ']': case '}':
            right = *pars->input++; /* pass the bracket so we don't see it again */
            if (is_first) {
                synerror(parsource(pars), "unexpected right bracket %c", right);
            } else if (left == '\'') {
                synerror(parsource(pars), "quote ' followed by right bracket %c",
                         right);
            } else if (!brackets_match(left, right)) {
                synerror(parsource(pars), "%c does not match %c", right, left);
            } else {
                return NULL;
            }
        case '{':
            pars->input++;
            synerror(parsource(pars), "curly brackets are not supported");
        default:
            if (read_tick_as_quote && *pars->input == '\'') {
                ⟨read a Par and return that Par wrapped in quote S171c⟩
            } else {
                ⟨read and return an ATOM S171d⟩
            }
        }
    }
}
```

With this code in hand, getpar is a first call.

S171a. ⟨*lex.c* S169c⟩+≡ ◁S170 S172a▷

```
Par getpar(Parstream pars) {
    assert(pars);
    return getpar_in_context(pars, true, '\0');
}
```

To scan past whitespace, getpar_in_context uses the standard C library function isspace. That function requires an *unsigned* character.

S171b. ⟨*advance* pars->input *past whitespace characters* S171b⟩≡ (S170)

```
while (isspace((unsigned char)*pars->input))
    pars->input++;
```

When getpar_in_context sees a quote mark “',” if it is reading a language that uses a ' operator, it reads the next Par (for example, (1 2 3)) and then returns that Par wrapped in quote (for example, (quote (1 2 3))).

S171c. ⟨*read a* Par *and return that* Par *wrapped in* quote S171c⟩≡ (S170)

```
{
    pars->input++;
    Par p = getpar_in_context(pars, false, '\'');
    if (p == NULL)
        synerror(parsource(pars), "premature end of file after quote mark");
    assert(p);
    return mkList(mkPL(mkAtom(strtoname("quote")), mkPL(p, NULL)));
}
```

Atoms are delegated to function readatom, defined below.

S171d. ⟨*read and return an* ATOM S171d⟩≡ (S170)

```
return mkAtom(readatom(&pars->input));
```

S171e. ⟨*prototypes of private functions that help with* getpar S171e⟩≡ (S170) S172b▷

```
static Name readatom(const char **ps);
```

Reading and returning a parenthesized list After it has seen a left parenthesis, getpar_in_context reads Pars until it sees a right parenthesis. It adds each Par to the front of elems_reversed. When it gets to the closing right parenthesis, it reverses the elements in place and returns the resulting list.

S171f. ⟨*read and return a parenthesized* LIST S171f⟩≡ (S170)

```
{
    char left = *pars->input++;   /* remember the opening left bracket */

    Parlist elems_reversed = NULL;
    Par q;  /* next par read in, to be accumulated into elems_reversed */
    while ((q = getpar_in_context(pars, false, left)))
        elems_reversed = mkPL(q, elems_reversed);

    if (pars->input == NULL)
        synerror(parsource(pars),
                 "premature end of file reading list (missing right parenthesis)");
    else
        return mkList(reverse_parlist(elems_reversed));
}
```

The list of Pars is reversed using a classic trick of imperative programming: update the pointers in place. The invariant is exactly the same as the invariant of revapp in Section 2.3.2 (page 99). But the code in Section 2.3.2 allocates new memory, and this code doesn't. Just updating pointers is enough.

S172a. ⟨*lex.c* S169c⟩+≡ ◁S171a S172c▷

```
static Parlist reverse_parlist(Parlist p) {
    Parlist reversed = NULL;
    Parlist remaining = p;
    /* Invariant: reversed followed by reverse(remaining) equals reverse(p) */
    while (remaining) {
        Parlist next = remaining->tl;
        remaining->tl = reversed;
        reversed = remaining;
        remaining = next;
    }
    return reversed;
}
```

S172b. ⟨*prototypes of private functions that help with* getpar S171e⟩+≡ (S170) ◁S171e S173a▷

```
static Parlist reverse_parlist(Parlist p);
```

Reading and returning an atom A lexical analyzer consumes input one character at a time. My code works with a pointer to the input characters. A typical function uses such a pointer to look at the input, converts some of the input to a result, and *updates* the pointer to point to the remaining, unconsumed input. To make the update possible, I must pass a *pointer* to the pointer, which has type char **.[2] Here, for example, readatom consumes the characters that form a single atom.

S172c. ⟨*lex.c* S169c⟩+≡ ◁S172a S172d▷

```
static Name readatom(const char **ps) {
    const char *p, *q;

    p = *ps;                            /* remember starting position */
    for (q = p; !isdelim(*q); q++)      /* scan to next delimiter */
        ;
    *ps = q;                            /* unconsumed input starts with delimiter */
    return strntoname(p, q - p);        /* the name is the difference */
}
```

A *delimiter* is a character that marks the end of a name or a token. In bridge languages, delimiters include parentheses, semicolon, whitespace, and end of string.

S172d. ⟨*lex.c* S169c⟩+≡ ◁S172c S172e▷

```
static int isdelim(char c) {
    return c == '(' || c == ')' || c == '[' || c == ']' || c == '{' || c == '}' ||
           c == ';' || isspace((unsigned char)c) ||
           c == '\0';
}
```

Function strntoname returns a name built from the first n characters of a string.

S172e. ⟨*lex.c* S169c⟩+≡ ◁S172d S173b▷

```
static Name strntoname(const char *s, int n) {
    char *t = malloc(n + 1);
    assert(t != NULL);
    strncpy(t, s, n);
    t[n] = '\0';
    return strtoname(t);
}
```

[2] In C++, I would instead pass the pointer by reference.

S173a. ⟨*prototypes of private functions that help with* `getpar` S171e⟩+≡ (S170) ◁ S172b S173c ▷

```
static int  isdelim(char c);
static Name strntoname(const char *s, int n);
```

S173b. ⟨*lex.c* S169c⟩+≡ ◁ S172e

```
static bool brackets_match(char left, char right) {
    switch (left) {
        case '(': return right == ')';
        case '[': return right == ']';
        case '{': return right == '}';
        default: assert(0);
    }
}
```

S173c. ⟨*prototypes of private functions that help with* `getpar` S171e⟩+≡ (S170) ◁ S173a

```
static bool brackets_match(char left, char right);
```

F.2.3 Streams of extended definitions

Layered on top of a `Parstream` is an `XDefstream`. One `Par` in the input corresponds exactly to one `XDef`, so the only state needed in an `XDefstream` is the `Parstream` it is made from.

S173d. ⟨*xdefstream.c* S173d⟩≡ S173e ▷

```
struct XDefstream {
    Parstream pars;                  /* where input comes from */
};
```

To make an `XDefstream`, allocate and initialize.

S173e. ⟨*xdefstream.c* S173d⟩+≡ ◁ S173d S173f ▷

```
XDefstream xdefstream(Parstream pars) {
    XDefstream xdefs = malloc(sizeof(*xdefs));
    assert(xdefs);
    assert(pars);
    xdefs->pars = pars;
    return xdefs;
}
```

The code in Chapter 1 doesn't even know that `Parstream`s exist. It builds `XDefstream`s by calling `filexdefs` or `stringxdefs`. In their turn, those functions build `XDefstream`s by combining `xdefstream` and `parstream` with either `filelines` or `stringlines`, respectively.

S173f. ⟨*xdefstream.c* S173d⟩+≡ ◁ S173e S173g ▷

```
XDefstream filexdefs(const char *filename, FILE *input, Prompts prompts) {
    return xdefstream(parstream(filelines(filename, input), prompts));
}
XDefstream stringxdefs(const char *stringname, const char *input) {
    return xdefstream(parstream(stringlines(stringname, input), NOT_PROMPTING));
}
```

To get an extended definition from an `XDefstream`, get a `Par` and parse it. The heavy lifting is done by `parsexdef`, which is the subject of Appendix G.

S173g. ⟨*xdefstream.c* S173d⟩+≡ ◁ S173f

```
XDef getxdef(XDefstream xdr) {
    Par p = getpar(xdr->pars);
    if (p == NULL)
        return NULL;
    else
        return parsexdef(p, parsource(xdr->pars));
}
```

F.3 Buffering characters

Alas, when `runerror` detects a run-time error, it's not OK for it to write a message to standard error: if `runerror` is called during a `check-error` test, then the test passes as expected, and no output should be written. To control the output from runerror, the error message is written into a resizeable *buffer*, by function `bprint` or `vbprint` (chunks S176d and S177c). The contents are printed only when the dynamic context warrants it. The buffer abstraction has type `Printbuf`, and it is declared and implemented here.

S174a. ⟨*shared type definitions* S165a⟩+≡ (S295a) ◁S168d S177a▷

```
typedef struct Printbuf *Printbuf;
```

A buffer is created with `printbuf` and destroyed with `freebuf`.

S174b. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S169a S174c▷

```
Printbuf printbuf(void);
void freebuf(Printbuf *);
```

A buffer can be appended to by `bufput` or `bufputs`; it is emptied by `bufreset`.

S174c. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S174b S174d▷

```
void bufput(Printbuf, char);
void bufputs(Printbuf, const char*);
void bufreset(Printbuf);
```

The contents of a buffer can be copied to a freshly allocated block of memory, or they can be written to an open file handle.

S174d. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S174c S176d▷

```
char *bufcopy(Printbuf);
void fwritebuf(Printbuf buf, FILE *output);
```

F.3.1 Implementation of a print buffer

This classic data structure should need no introduction.

S174e. ⟨*printbuf.c* S174e⟩≡ S174f▷

```
struct Printbuf {
    char *chars;  // start of the buffer
    char *limit;  // marks one past end of buffer
    char *next;   // where next character will be buffered
    // invariants: all are non-NULL
    //             chars <= next <= limit
    //             if chars <= p < limit, then *p is writeable
};
```

A buffer initially holds 100 characters.

S174f. ⟨*printbuf.c* S174e⟩+≡ ◁S174e S175a▷

```
Printbuf printbuf(void) {
   Printbuf buf = malloc(sizeof(*buf));
   assert(buf);
   int n = 100;
   buf->chars = malloc(n);
   assert(buf->chars);
   buf->next  = buf->chars;
   buf->limit = buf->chars + n;
   return buf;
}
```

A buffer is freed using Hanson's (1996) indirection trick.

S175a. ⟨*printbuf.c* S174e⟩+≡ ◁S174f S175b▷

```
void freebuf(Printbuf *bufp) {
   Printbuf buf = *bufp;
   assert(buf && buf->chars);
   free(buf->chars);
   free(buf);
   *bufp = NULL;
}
```

Calling grow makes a buffer 30% larger, or at least 1 byte larger.

S175b. ⟨*printbuf.c* S174e⟩+≡ ◁S175a S175c▷

```
static void grow(Printbuf buf) {
    assert(buf && buf->chars && buf->next && buf->limit);
    unsigned n = buf->limit - buf->chars;
    n = 1 + (n * 13) / 10;   // 30% size increase
    unsigned i = buf->next - buf->chars;
    buf->chars = realloc(buf->chars, n);
    assert(buf->chars);
    buf->next  = buf->chars + i;
    buf->limit = buf->chars + n;
}
```

Function `bufput` writes one character to `buf->next`, growing the buffer when needed.

S175c. ⟨*printbuf.c* S174e⟩+≡ ◁S175b S175d▷

```
void bufput(Printbuf buf, char c) {
    assert(buf && buf->next && buf->limit);
    if (buf->next == buf->limit) {
        grow(buf);
        assert(buf && buf->next && buf->limit);
        assert(buf->limit > buf->next);
    }
    *buf->next++ = c;
}
```

Function `bufputs`, which writes an entire string, first ensures that the buffer is large enough to hold the string, then copies the string by calling `memcpy`.

S175d. ⟨*printbuf.c* S174e⟩+≡ ◁S175c S175e▷

```
void bufputs(Printbuf buf, const char *s) {
    assert(buf);
    int n = strlen(s);
    while (buf->limit - buf->next < n)
        grow(buf);
    memcpy(buf->next, s, n);
    buf->next += n;
}
```

Function `bufreset` discards all the characters.

S175e. ⟨*printbuf.c* S174e⟩+≡ ◁S175d S176a▷

```
void bufreset(Printbuf buf) {
    assert(buf && buf->next);
    buf->next = buf->chars;
}
```

To use the buffer, client code may want to know how many characters are in it.

S176a. ⟨*printbuf.c* S174e⟩+≡ ◁S175e S176b▷

```
static int nchars(Printbuf buf) {
    assert(buf && buf->chars && buf->next);
    return buf->next - buf->chars;
}
```

Function `bufcopy` copies a buffer's contents to a fresh block.

S176b. ⟨*printbuf.c* S174e⟩+≡ ◁S176a S176c▷

```
char *bufcopy(Printbuf buf) {
   assert(buf);
   int n = nchars(buf);
   char *s = malloc(n+1);
   assert(s);
   memcpy(s, buf->chars, n);
   s[n] = '\0';
   return s;
}
```

And function `fwritebuf` writes a buffer's characters to an open file handle.

S176c. ⟨*printbuf.c* S174e⟩+≡ ◁S176b

```
void fwritebuf(Printbuf buf, FILE *output) {
    assert(buf && buf->chars && buf->limit);
    assert(output);
    int n = fwrite(buf->chars, sizeof(*buf->chars), nchars(buf), output);
    assert(n == nchars(buf));
}
```

F.4 The extensible buffer printer

To recapitulate Section 1.6.1, the standard C functions `printf` and `fprintf` are great, but they don't know how to print things like values and expressions. And when you can't put a value or an expression in a format string, the code needed to print an error message becomes awkward and unreadable. To enable simple, readable printing code, I define new, custom print functions that know how to print values and expressions:

S176d. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S174d S176e▷

```
void print (const char *fmt, ...);              // print to standard output
void fprint(FILE *output, const char *fmt, ...);    // print to given file
void bprint(Printbuf output, const char *fmt, ...); // print to given buffer
```

I use `bprint` to write error messages—if an error message is written during the evaluation of a `check-expect` or `check-error`, the message can be captured and can either be used to explain what went wrong (if an error occurs unexpectedly during a `check-expect`) or can be silently discarded (if an error occurs as expected during a `check-error`).

Dealing with a variable number of arguments is a hassle, and I may as well do it only once. So I don't just define a couple of print functions that know about values and expressions in one language. Instead, I make them *extensible,* so they can deal with any language.

To extend a printer, client code calls `installprinter` to announce a new format specifier. The call receives a function that will be used to print a value so specified.

S176e. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S176d S177c▷

```
void installprinter(unsigned char specifier, Printer *take_and_print);
```

The function provided has type Printer. When called, it takes one value out of the list args, then prints the value to the given buffer.

S177a. ⟨*shared type definitions* S165a⟩+≡ (S295a) ◁S174a

```
⟨definition of va_list_box S177b⟩
typedef void Printer(Printbuf output, va_list_box *args);
```

The type va_list_box is almost, but not quite, a standard C type for holding a variable number of arguments. A function that can accept a variable number of arguments is called *variadic,* and according to the C standard, the arguments of a variadic function are stored in an object of type va_list, which is defined in the standard library in header file stdarg.h. (If you are not accustomed to variadic functions and stdarg.h, you may wish to consult Sections 7.2 and 7.3 of Kernighan and Ritchie 1988.) So what is va_list_box? It's a workaround for a bug that afflicts some versions of the GNU C compiler on 64-bit hardware. These compilers fail when values of type va_list are passed as arguments.[3] A workaround for this problem is to place the va_list in a structure and pass a pointer to the structure. That structure is called va_list_box, and it is defined here:

S177b. ⟨*definition of* va_list_box S177b⟩≡ (S177a)

```
typedef struct va_list_box {
  va_list ap;
} va_list_box;
```

Think of the printing infrastructure as a stack of bricks:

- There are two foundation bricks: the buffer abstraction defined in the previous section, and the C standard machinery for defining variadic functions: header file stdarg.h, type va_list, and macros va_start, va_arg, and va_end. Many C programmers haven't studied this machinery, and if you're among them, you'll want either to review it or to skip this section.

- The next brick is my function vbprint and its associated table printertab. Function vbprint stands in the same relation to bprint as standard function vfprintf stands to fprintf:

  S177c. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S176e S179c▷

  ```
  void vbprint(Printbuf output, const char *fmt, va_list_box *box);
  ```

  The printertab table, which is private to the printing module, associates a Printer function to each possible conversion specifier. This style of programming exploits first-class functions in C, drawing on some of the ideas presented as part of μScheme in Chapter 2. Function installprinter simply updates printertab.

- The next bricks define bprint, print, and fprint on top of vbprint.

- The final bricks each have type Printfun. There are a whole bunch of them: one for each conversion specifier we know how to print (there's a list in Table 1.6, page 47).

type Printbuf S174a

In Section F.5.1 (page S182), these same bricks also support functions runerror, othererror, and synerror.

None of the ideas here are new; extensible printers have long popular with sophisticated C programmers. If you want to study an especially well-crafted example, consult Hanson (1996, chapter 14).

[3] Library functions such as vfprintf itself are OK; only *users* cannot write functions that take va_list arguments. Feh.

F.4.1 Building variadic functions on top of `vbprint`

Function `bprint` is a wrapper around `vbprint`. It calls `va_start` to initialize the list of arguments in `box`, passes the arguments to `vbprint`, and calls `va_end` to finalize the arguments. The calls to `va_start` and `va_end` are mandated by the C standard.

S178a. ⟨*print.c* S178a⟩≡ S178b ▷

```
void bprint(Printbuf output, const char *fmt, ...) {
    va_list_box box;

    assert(fmt);
    va_start(box.ap, fmt);
    vbprint(output, fmt, &box);
    va_end(box.ap);
}
```

Function `print` buffers, then prints. It keeps a buffer in a cache.

S178b. ⟨*print.c* S178a⟩+≡ ◁ S178a S178c ▷

```
void print(const char *fmt, ...) {
    va_list_box box;
    static Printbuf stdoutbuf;

    if (stdoutbuf == NULL)
        stdoutbuf = printbuf();

    assert(fmt);
    va_start(box.ap, fmt);
    vbprint(stdoutbuf, fmt, &box);
    va_end(box.ap);
    fwritebuf(stdoutbuf, stdout);
    bufreset(stdoutbuf);
    fflush(stdout);
}
```

Function `fprint` caches its own buffer.

S178c. ⟨*print.c* S178a⟩+≡ ◁ S178b S179a ▷

```
void fprint(FILE *output, const char *fmt, ...) {
    static Printbuf buf;
    va_list_box box;

    if (buf == NULL)
        buf = printbuf();

    assert(fmt);
    va_start(box.ap, fmt);
    vbprint(buf, fmt, &box);
    va_end(box.ap);
    fwritebuf(buf, output);
    fflush(output);
    freebuf(&buf);
}
```

F.4.2 Implementations of `vbprint` *and* `installprinter`

Function `vbprint`'s primary job is to decode the format string and to find all the conversion specifiers. Each time it sees a conversion specifier, it calls the corresponding Printer. The Printer for a conversion specifier c is stored in printertab[(unsigned char)c].

S179a. ⟨*print.c* S178a⟩+≡ ◁S178c S179b▷

```
static Printer *printertab[256];

void vbprint(Printbuf output, const char *fmt, va_list_box *box) {
    const unsigned char *p;
    bool broken = false; // made true on seeing an unknown conversion specifier
    for (p = (const unsigned char*)fmt; *p; p++) {
        if (*p != '%') {
            bufput(output, *p);
        } else {
            if (!broken && printertab[*++p])
                printertab[*p](output, box);
            else {
                broken = true;  /* box is not consumed */
                bufputs(output, "<pointer>");
            }
        }
    }
}
```

The va_arg interface is unsafe, and if a printing function takes the wrong thing from box, a memory error could ensue. So if vbprint ever sees a conversion specifier that it doesn't recognize, it stops calling printing functions.

Function installprinter simply stores to the private table.

S179b. ⟨*print.c* S178a⟩+≡ ◁S179a

```
void installprinter(unsigned char c, Printer *take_and_print) {
    printertab[c] = take_and_print;
}
```

F.4.3 Printing functions

The most interesting printing functions are language-dependent; they are found in Appendices K and L. But functions that print percent signs, strings, decimal integers, characters, and names are shared among all languages, and they are found here.

S179c. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S177c S180e▷

```
Printer printpercent, printstring, printdecimal, printchar, printname,
        printpointer;
```

As in standard vprintf, the conversion specifier %% just prints a percent sign, without consuming any arguments.

S179d. ⟨*printfuns.c* S179d⟩≡ S179e▷

```
void printpercent(Printbuf output, va_list_box *box) {
    (void)box;
    bufput(output, '%');
}
```

The printers for strings, pointers, and numbers are textbook examples of how to use va_arg. (Hey! You are reading a textbook!)

S179e. ⟨*printfuns.c* S179d⟩+≡ ◁S179d S180a▷

```
void printstring(Printbuf output, va_list_box *box) {
    const char *s = va_arg(box->ap, char*);
    bufputs(output, s);
}
```

S180a. ⟨*printfuns.c* S179d⟩+≡ ◁S179e S180b▷

```
void printdecimal(Printbuf output, va_list_box *box) {
    char buf[2 + 3 * sizeof(int)];
    snprintf(buf, sizeof(buf), "%d", va_arg(box->ap, int));
    bufputs(output, buf);
}
```

S180b. ⟨*printfuns.c* S179d⟩+≡ ◁S180a S180c▷

```
void printpointer(Printbuf output, va_list_box *box) {
    char buf[12 + 3 * sizeof(void *)];
    snprintf(buf, sizeof(buf), "%p", va_arg(box->ap, void *));
    bufputs(output, buf);
}
```

The printer for names prints a name's string. A `Name` should never be `NULL`, but if something goes drastically wrong and a `NULL` pointer is printed as a name, the code won't crash.

S180c. ⟨*printfuns.c* S179d⟩+≡ ◁S180b S180d▷

```
void printname(Printbuf output, va_list_box *box) {
    Name np = va_arg(box->ap, Name);
    bufputs(output, np == NULL ? "<null>" : nametostr(np));
}
```

S180d. ⟨*printfuns.c* S179d⟩+≡ ◁S180c S180f▷

```
void printchar(Printbuf output, va_list_box *box) {
    int c = va_arg(box->ap, int);
    bufput(output, c);
}
```

The print function for parenthesized phrases is surprisingly simple: it just calls `bprint` recursively:

S180e. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S179c S181a▷

```
Printer printpar;
```

S180f. ⟨*printfuns.c* S179d⟩+≡ ◁S180d

```
void printpar(Printbuf output, va_list_box *box) {
    Par p = va_arg(box->ap, Par);
    if (p == NULL) {
        bprint(output, "<null>");
        return;
    }

    switch (p->alt){
    case ATOM:
        bprint(output, "%n", p->atom);
        break;
    case LIST:
        bprint(output, "(%P)", p->list);
        break;
    }
}
```

The `%P` specifier is associated with function `printparlist`, which is generated automatically by the same script that generates all the list codes. The automatically generated code looks something like this:

```
void printparlist(Printbuf output, va_list_box *box) {
    for (Parlist ps = va_arg(box->ap, Parlist); ps != NULL; ps = ps->tl)
        bprint(output, "%p%s", ps->hd, ps->tl ? " " : "");
}
```

F.5 Error functions

The interface in Section 1.6.1 (page 46) shows functions `runerror` and `synerror`, which behave a lot like `bprint`, but which, after buffering, `longjmp` to the `jmp_buf` `errorjmp`. To understand Chapter 1, that's all you need to know, but there's more to the story. When running a unit test, the error infrastructure must *not* print messages or transfer control to `errorjmp`. When a run-time error occurs, a unit test mustn't print a standard message or return control to the read-eval-print loop. Instead, it must know that the error has occurred so that it can decide what the error means: does the unit test pass (`check-error`) or fail (`check-expect`)? For unit testing, I therefore provide a second, *testing* mode in which the error-signaling functions can operate.

In testing mode, `runerror` buffers an error message and `longjmp`s to `testjmp`.

S181a. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S180e S184a▷

```
typedef enum ErrorMode { NORMAL, TESTING } ErrorMode;
void set_error_mode(ErrorMode mode);
extern jmp_buf testjmp;    // if error occurs during a test, longjmp here
extern Printbuf errorbuf;  // if error occurs during a test, message is here
```

The error mode is initially `NORMAL`, but it can be changed using `set_error_mode`. When the error mode is changed to `TESTING`, it is an unchecked run-time error to call `synerror`, and it is an unchecked run-time error to call `runerror` except while a `setjmp` involving `testjmp` is active on the C call stack.

F.5.1 Implementation of error signaling

The state of the error module includes the error mode and the two `jmp_buf`s.

S181b. ⟨*error.c* S181b⟩≡ S181c▷

```
jmp_buf errorjmp;
jmp_buf testjmp;

static ErrorMode mode = NORMAL;
```

The error mode is set by Function `set_error_mode`.

S181c. ⟨*error.c* S181b⟩+≡ ◁S181b S182▷

```
void set_error_mode(ErrorMode new_mode) {
  assert(new_mode == NORMAL || new_mode == TESTING);
  mode = new_mode;
}
```

`bprint`	S176d
`bufput`	S174c
`bufputs`	S174c
`type Name`	43a
`nametostr`	43b
`type Par`	$\mathcal{A}$
`type Printbuf`	S174a
`type Printer`	S177a
`type va_list_box`	S177b

The mode determines how function `runerror` behaves:

- In normal mode, `runerror` prints a message, then jumps to `errorjmp`.
- In testing mode, `runerror` *buffers* the message, then silently jumps to `testjmp`.

S182. ⟨*error.c* S181b⟩+≡ ◁S181c S183a▷

```
Printbuf errorbuf;
void runerror(const char *fmt, ...) {
    va_list_box box;

    if (!errorbuf)
        errorbuf = printbuf();

    assert(fmt);
    va_start(box.ap, fmt);
    vbprint(errorbuf, fmt, &box);
    va_end(box.ap);

    switch (mode) {
    case NORMAL:
        fflush(stdout);
        char *msg = bufcopy(errorbuf);
        fprintf(stderr, "Run-time error: %s\n", msg);
        fflush(stderr);
        free(msg);
        bufreset(errorbuf);
        longjmp(errorjmp, 1);

    case TESTING:
        longjmp(testjmp, 1);

    default:
        assert(0);
    }
}
```

Function `synerror` is like `runerror`, but with additional logic for printing source-code locations. Source-code locations are printed *except* from standard input in the `WITHOUT_LOCATIONS` mode.

S183a. ⟨*error.c* S181b⟩+≡ ◁S182 S183b▷

```
static ErrorFormat toplevel_error_format = WITH_LOCATIONS;

void synerror(Sourceloc src, const char *fmt, ...) {
    va_list_box box;

    switch (mode) {
    case NORMAL:
        assert(fmt);
        fflush(stdout);
        if (toplevel_error_format == WITHOUT_LOCATIONS
        && !strcmp(src->sourcename, "standard input"))
            fprint(stderr, "syntax error: ");
        else
            fprint(stderr, "syntax error in %s, line %d: ", src->sourcename, src->line);
        Printbuf buf = printbuf();
        va_start(box.ap, fmt);
        vbprint(buf, fmt, &box);
        va_end(box.ap);

        fwritebuf(buf, stderr);
        freebuf(&buf);
        fprintf(stderr, "\n");
        fflush(stderr);
        longjmp(errorjmp, 1);

    default:
        assert(0);
    }
}
```

Function `set_toplevel_error_format` sets the error format used for standard input.

S183b. ⟨*error.c* S181b⟩+≡ ◁S183a S184b▷

```
void set_toplevel_error_format(ErrorFormat new_format) {
  assert(new_format == WITH_LOCATIONS || new_format == WITHOUT_LOCATIONS);
  toplevel_error_format = new_format;
}
```

`bufcopy`	S174d
`bufreset`	S174c
type `ErrorFormat`	S294a
`errorjmp`	S181b
`fprint`	S176d
`freebuf`	S174b
`fwritebuf`	S174d
`mode`	S181b
type `Printbuf`	S174a
`printbuf`	S174b
type `Sourceloc`	S293h
`testjmp`	S181b
type `va_list_box`	S177b
`vbprint`	S177c

Function othererror generalizes runerror. It's used only in Chapter 3, to signal if an error occurs while μScheme+ is being lowered to Core μScheme+.

S184a. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁ S181a S184d ▷

```
void othererror (const char *fmt, ...);
```

S184b. ⟨*error.c* S181b⟩+≡ ◁ S183b S184c ▷

```
void othererror(const char *fmt, ...) {
    va_list_box box;

    if (!errorbuf)
        errorbuf = printbuf();

    assert(fmt);
    va_start(box.ap, fmt);
    vbprint(errorbuf, fmt, &box);
    va_end(box.ap);

    switch (mode) {
    case NORMAL:
        fflush(stdout);
        char *msg = bufcopy(errorbuf);
        fprintf(stderr, "%s\n", msg);
        fflush(stderr);
        free(msg);
        bufreset(errorbuf);
        longjmp(errorjmp, 1);

    case TESTING:
        longjmp(testjmp, 1);

    default:
        assert(0);
    }
}
```

F.5.2 Implementations of error helpers

As promised in Section 1.6.1 (page 47), this appendix implements auxiliary functions that help detect common errors. Function checkargc checks to see if the number of actual arguments passed to a function is the number that the function expected.

S184c. ⟨*error.c* S181b⟩+≡ ◁ S184b S185a ▷

```
void checkargc(Exp e, int expected, int actual) {
    if (expected != actual)
        runerror("in %e, expected %d argument%s but found %d",
                 e, expected, expected == 1 ? "" : "s", actual);
}
```

If a list of names contains duplicates, duplicatename returns a duplicate. It is used to detect duplicate names in lists of formal parameters. Its cost is quadratic in the number of parameters, which for any reasonable function, should be very fast.

S184d. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁ S184a S186a ▷

```
Name duplicatename(Namelist names);
```

S185a. ⟨*error.c* S181b⟩+≡ ◁ S184c

```
Name duplicatename(Namelist xs) {
    if (xs != NULL) {
        Name n = xs->hd;
        for (Namelist tail = xs->tl; tail; tail = tail->tl)
            if (n == tail->hd)
                return n;
        return duplicatename(xs->tl);
    }
    return NULL;
}
```

The tail call could be turned into a loop, but it hardly seems worth it. (Quirks of the C standard prevent C compilers from optimizing all tail calls, but a good C compiler will identify and optimize a direct tail recursion like this one.)

F.6 Test processing and reporting

Code that runs unit tests has to call `process_test`, which is language-dependent. That code is found in Appendices K and L. But the code that reports the results is language-independent and is found here:

S185b. ⟨*tests.c* S185b⟩≡

```
void report_test_results(int npassed, int ntests) {
    switch (ntests) {
    case 0: break; /* no report */
    case 1:
        if (npassed == 1)
            printf("The only test passed.\n");
        else
            printf("The only test failed.\n");
        break;
    case 2:
        switch (npassed) {
        case 0: printf("Both tests failed.\n"); break;
        case 1: printf("One of two tests passed.\n"); break;
        case 2: printf("Both tests passed.\n"); break;
        default: assert(0); break;
        }
        break;
    default:
        if (npassed == ntests)
            printf("All %d tests passed.\n", ntests);
        else if (npassed == 0)
            printf("All %d tests failed.\n", ntests);
        else
            printf("%d of %d tests passed.\n", npassed, ntests);
        break;
    }
}
```

`bufcopy`	S174d
`bufreset`	S174c
`errorbuf`	S182
`errorjmp`	S181b
type `Exp`	$\mathcal{A}$
`mode`	S181b
type `Name`	43a
type `Namelist`	43a
`printbuf`	S174b
`runerror`	47a
`testjmp`	S181b
type `va_list_box`	S177b
`vbprint`	S177c

F.7 Stack-overflow detection

If somebody writes a recursive Impcore or μScheme function that calls itself forever, what should the interpreter do? An ordinary recursive `eval` would call *itself* forever, and eventually the C code would run out of resources and would be terminated. There's a better way. My implementation of `eval` contains a hidden call to a function called `checkoverflow`, which detects very deep recursion and calls `runerror`.

The implementation uses C trickery with `volatile` variables: the address of a `volatile` local variable `c` is used as a proxy for the stack pointer. (Because I spent years writing compilers, I understand a little of how these things work.) The first call to `checkoverflow` captures the stack pointer and stores as a “low-water mark.” Each later call checks the current stack pointer against that low-water mark. If the distance exceeds `limit`, `checkoverflow` calls `runerror`. Otherwise it returns the distance.

S186a. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S184d S187b▷

```
extern int  checkoverflow(int limit);
extern void reset_overflow_check(void);
```

I assume that the stack grows downward.

S186b. ⟨*overflow.c* S186b⟩≡

```
static volatile char *low_water_mark = NULL;

#define N 600 // fuel in units of 10,000

static int default_eval_fuel = N * 10000;
static int eval_fuel         = N * 10000;
static bool throttled = 1;
static bool env_checked = 0;

int checkoverflow(int limit) {
  volatile char c;
  if (!env_checked) {
      env_checked = 1;
      const char *options = getenv("BPCOPTIONS");
      if (options == NULL)
          options = "";
      throttled = strstr(options, "nothrottle") == NULL;
  }
  if (low_water_mark == NULL) {
    low_water_mark = &c;
    return 0;
  } else if (low_water_mark - &c >= limit) {
    runerror("recursion too deep");
  } else if (throttled && eval_fuel-- <= 0) {
    eval_fuel = default_eval_fuel;
    runerror("CPU time exhausted");
  } else {
    return (low_water_mark - &c);
  }
}

extern void reset_overflow_check(void) {
  eval_fuel = default_eval_fuel;
}
```

When a stack overflow is detected, it manifests like this:

S187a. ⟨*transcript* S187a⟩≡ S187e ▷

```
-> (define blowstack (n) (+ 1 (blowstack (- n 1))))
-> (blowstack 0)
Run-time error: recursion too deep
```

F.8 Arithmetic-overflow detection

Unlike standard C arithmetic, the arithmetic in this book detects *arithmetic overflow*: an operation on 32-bit signed integers whose result cannot also be represented as a 32-bit signed integer. Such arithmetic is defined by the C standard as "undefined behavior," so my code needs to detect it before it might happen. Function `checkarith` does arithmetic using 64-bit integers, and if the result does not fit in the specified number of bits, it triggers a checked run-time error.

S187b. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁ S186a S188a ▷

```
extern void checkarith(char operation, int32_t n, int32_t m, int precision);
```

Only addition, subtraction, multiplication, and division can cause overflow.

S187c. ⟨*arith.c* S187c⟩≡

```
void checkarith(char operation, int32_t n, int32_t m, int precision) {
  int64_t nx = n;
  int64_t mx = m;
  int64_t result;
  switch (operation) {
    case '+': result = nx + mx; break;
    case '-': result = nx - mx; break;
    case '*': result = nx * mx; break;
    case '/': result = mx != 0 ? nx / mx : 0; break;
    default:  return;  /* other operations can't overflow */
  }
  ⟨if result cannot be represented using precision signed bits, signal overflow S187d⟩
}
```

A 64-bit `result` fits in k bits if it is unchanged by sign-extending the least significant k bits. Sign extension is achieved by two shifts. According to the C standard, shifts on `int64_t` are defined up to 63 bits.

S187d. ⟨*if* `result` *cannot be represented using* `precision` *signed bits, signal overflow* S187d⟩≡ (S187c)

```
assert(precision > 0 && precision < 64);  // shifts are defined
if ((result << (64-precision)) >> precision != result) {
  runerror("Arithmetic overflow");
}
```

When reported to a user, arithmetic overflow looks like this:

S187e. ⟨*transcript* S187a⟩+≡ ◁ S187a

```
-> (define one-bits (n) (if (= n 0) 0 (+ 1 (* 2 (one-bits (- n 1))))))
-> (one-bits 30)
1073741823
-> (one-bits 31)
2147483647
-> (one-bits 32)
Run-time error: Arithmetic overflow
```

`runerror`	47a

F.9 Unicode support

Unicode is a standard that attempts to describe all the world's character sets. In Unicode, each character is described by a "code point," which is an unsigned integer. Example code points include "capital A" (code point 65) and "capital Å with a circle over it" (code point 197). Most character sets fit in the *Basic Multilingual Plane*, whose code points can be expressed as 16-bit unsigned integers.

UTF-8 stands for "Unicode Transfer Format (8 bits)." UTF-8 is a *variable-length binary code* in which each 16-bit code point is coded as a one-byte, two-byte, or three-byte *UTF-8 sequence*. The coding of code points with values up to 65535 is as follows:

```
hex         binary                     UTF-8 binary
0000-007F   00000000 0abcdefg   =>   0abcdefg
0080-07FF   00000abc defghijk   =>   110abcde 10fghijk
0800-FFFF   abcdefgh ijklmnop   =>   1110abcd 10efghij 10klmnop

010000-001FFFFF:      11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

Code points from Western languages have short UTF-8 sequences: often one byte, almost always two.

Unicode characters are printed as UTF-8 sequences by these two functions:

S188a. ⟨*shared function prototypes* S165b⟩+≡ (S295a) ◁S187b

```
void fprint_utf8(FILE *output, unsigned code_point);
void print_utf8 (unsigned u);
```

This encoder supports code points of up to 21 bits.

S188b. ⟨*unicode.c* S188b⟩≡ S188c ▷

```
void fprint_utf8(FILE *output, unsigned code_point) {
    if ((code_point & 0x1fffff) != code_point)
        runerror("%d does not represent a Unicode code point", (int)code_point);
    if (code_point > 0xffff) {     // 21 bits
        putc(0xf0 |  (code_point >> 18),        output);
        putc(0x80 | ((code_point >> 12) & 0x3f), output);
        putc(0x80 | ((code_point >>  6) & 0x3f), output);
        putc(0x80 | ((code_point      ) & 0x3f), output);
    } else if (code_point > 0x7ff) { // 16 bits
        putc(0xe0 | (code_point >> 12),         output);
        putc(0x80 | ((code_point >> 6) & 0x3f), output);
        putc(0x80 | ((code_point     ) & 0x3f), output);
    } else if (code_point > 0x7f) { // 12 bits
        putc(0xc0 | (code_point >> 6),         output);
        putc(0x80 | (code_point & 0x3f),       output);
    } else {                        // 7 bits
        putc(code_point, output);
    }
}
```

S188c. ⟨*unicode.c* S188b⟩+≡ ◁S188b

```
void print_utf8(unsigned code_point) {
    fprint_utf8(stdout, code_point);
}
```

Appendix G contents

G
Parsing parenthesized phrases (including Impcore) in C

A key step in the implementation of any programming language is to translate the concrete syntax that appears in the input to the abstract syntax that is used internally. This translation is typically implemented in two steps: *lexical analysis* groups related characters into *tokens*, and *parsing* translates a sequence of tokens into one or more abstract-syntax trees. In the second part of this book, starting with Chapter 5, interpreters are written in Standard ML, and they follow exactly this model. But in the first part, where interpreters are written in C, follows a different model: sequences of lines are turned into *parenthesized phrases* (Section F.2.2), and these phrases are what is parsed into abstract syntax. The details are the subject of this appendix.

Considering what I hope you'll get out of this book, the implementation of a parser is a side issue. Parsing is an art and a science all its own, and it is the subject of its own learned textbooks. Using parenthesized phrases enables me to avoid the usual challenges and complexities. In their place, however, I have one challenge that is central to what I hope you get out of this book—to get the most out of the Exercises, you have to be able to add new syntactic forms. Using the parser I describe below, adding new syntactic forms is relatively easy: you add new entries to a couple of tables and a new case to a `switch` statement in a syntax-building function. But there is a cost: there's a lot of infrastructure to understand. Infrastructure is easier to understand if you can see how it's used, so along with the general parsing infrastructure, I present the code used to parse Impcore. But if you want to avoid studying infrastructure and just get on with adding new syntax, jump to the example and checklist in Section G.7 (page S209).

The parser in this appendix is easy for you to extend, and it happens to be reasonably efficient, but regrettably, it is not simple. However, it is based on classic ideas developed by Knuth (1965), so if you study it, you will have a leg up on the "LR parsers" which so dominated the second half of the twentieth century.[1]

To make it as easy as possible for you to extend parsers, I've split the code into two files. File `tableparsing.c` contains code that can be reused. This file is not only part of the Impcore interpreter but also part of interpreters for μScheme and μScheme$+$. File `parse.c` contains code that is specific to the language being parsed (here, Impcore). File `tableparsing.c` is never modified; if you want to extend a language, you modify only code from `parse.c`.

[1] Given the severe memory constraints imposed by machines of the 1970s, LR-parser generators like Yacc and Bison were brilliant innovations. In the 21st century, we have memory to burn, and you are better off choosing a parsing technology that will enable you to spend more time getting work done and less time engineering your grammar. But I digress.

G.1 Planning an extensible parser

A parser is a function that is given a `Par` and builds an abstract-syntax tree, which it then returns. Each of the first three bridge languages (Impcore, μScheme, and μScheme+) has two major syntactic categories, which means two types of abstract-syntax trees, which means two parsers.

S192a. ⟨*shared function prototypes* S192a⟩≡ (S295a) S192b ▷

```
Exp  parseexp (Par p, Sourceloc source);
XDef parsexdef(Par p, Sourceloc source);
```

Each parser also takes a pointer to a source-code location, which it uses if it has to report an error.

A parser gets a parenthesized phrase of type `Par` and builds an abstract-syntax tree. In this appendix, I call the `Par` an *input* and the abstract-syntax tree a *component*. Components include all the elements that go into an abstract-syntax trees; in Impcore, a component can be a name, a list of names, an expression, or a list of expressions.

Parsing begins with a look at the input, which is either an `ATOM` or a `LIST` of `Par`s. And the interpretation of the input depends on whether the parser is looking for an `Exp` or an `XDef`.

- If the input is an `ATOM`, the parser must be looking for an expression (in Impcore, a `VAR` or `LITERAL` expression), and the job of making it into an `Exp` is given to function `exp_of_atom`, which is language-dependent.

  S192b. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁ S192a S195a ▷

  ```
  Exp exp_of_atom(Sourceloc loc, Name atom);
  ```

- If the input is a `LIST`, there are two possibilities: the first element of the list is a reserved word, or it's not.

  – A reserved word like `val` or `define` identifies the input as a true definition.

    A reserved word like `use` or `check-expect` identifies the input as an extended definition.

    A reserved word like `set` or `if` identifies the form as an expression.

  – If there's no reserved word, the input must be a function application. (Consult any grammar and you'll see there's no other choice.)

  The `LIST` inputs require all the technology.

Once the parser sees a keyword, it knows what it's looking for. Each keyword specifies the construction of a node in an abstract-syntax tree, and the remaining inputs in the list are parsed to build the children of that node. The specifications are shown in Tables G.1 and G.2. Lack of a keyword is also a specification; the final row in the expression table means "if you're looking for an expression and you don't see an expression keyword, the input must be a function application." In the extended-definition table, it means "if you're looking for an extended definition and you don't see an extended-definition keyword, the input must be a top-level expression."

A *parsing function* like `parseexp` or `parsexdef` is organized around the left-to-right conversion of `Par`s to components.

Parsing is organized around syntactic forms. Each syntactic form comes with its own form of abstract syntax, but they have a lot of structure in common. On the abstract side, each syntactic form has *components* and is created with a *build* function. For example, a `set` expression has two components (a name and an expression) and is built with `mkSet`. As another example, an `if` expression has three

Table G.1: Parsing table for Impcore expressions

Keyword	Code	Components
`set`	`SET`	*name*, *exp*
`if`	`IFX`	*exp*, *exp*, *exp*
`while`	`WHILEX`	*exp*, *exp*
`begin`	`BEGIN`	list of *exp*
—	`APPLY`	*name*, list of *exp*

Table G.2: Parsing table for Impcore extended definitions

Keyword	Code	Components
`val`	(not shown)	*name*, *exp*
`define`	(not shown)	*name*, (not shown), *exp*
`use`	(not shown)	*name*
`check-expect`	(not shown)	*exp*, *exp*
`check-assert`	(not shown)	*exp*
`check-error`	(not shown)	*exp*
—	(not shown)	*exp*

components, all of which are expressions, and is built with `mkIfx`. Each syntactic form is identified by a small-integer code, like `SET` or `IFX`.

On the concrete side, forms are a little more diverse.

- Some forms, like `VAR` or `LITERAL`, are written syntactically using a single atom.
- Most forms, including `SET` and `IF`, are written syntactically as a sequence of `Par`s wrapped in parentheses. And with one exception, the first of these `Par`s is a keyword, like `set` or `if`. The exception is the function-application form. (For the extended definitions, the exception is the the top-level expression form—a top-level expression may begin with a keyword, but it's a keyword that the extended-definition parser won't recognize.)

These properties help determine a plan:

type `Exp`	$\mathcal{A}$
type `Name`	43a
type `Par`	$\mathcal{A}$
type `Sourceloc`	S293h
type `XDef`	$\mathcal{A}$

1. There will be two parsers: one for expressions and one for extended definitions.
2. If a parser sees an atom, it must know what to do.
3. If a parser sees a parenthesized `Parlist`, it will consult a *table* of *rows*.
    - Each row knows how to parse one syntactic form. What does it mean "to know how to parse"? The row begins with a keyword that the parser should look for. The row also includes an integer code that identifies the form, and finally, the row lists the components of the form. To see some example rows, look at the parsing table for Impcore, in Table G.1.
    - A row matches an input `Parlist` if the row's keyword is equal to the first element of the `Parlist`. The parser proceeds through the rows looking for one that matches its input.

4. Once the parser finds the right row, it gets each component from the input `Parlist`, then checks to make sure there are no leftover inputs. Finally it passes the components and the integer code to a *reduce function*. Impcore uses two such functions: `reduce_to_exp` and `reduce_to_xdef`. Each of these functions takes a sequence of components and reduces it to a single node in an abstract-syntax tree. (The name `reduce` comes from *shift-reduce parsing*, which refers to a family of parsing techniques of which my parsers are members.)

I've designed the parsers to work this way so that you can easily add new syntactic forms. It's as simple as adding a row to a table and a case to a reduce function. In more detail,

1. Decide whether you wish to add an expression form or a definition form. That will tell you what table and reduce function to modify. For example, if you want to add a new expression form, modify functions `exptable` and `reduce_to_exp`.

2. Choose a keyword and an unused integer code. As shown below, codes for extended definitions have to be chosen with a little care.

3. Add a row to your chosen table.

4. Add a case to your chosen reduce function.

I think you'll like being able to extend languages so easily, but there's a cost—the table-driven parser needs a lot of infrastructure. That infrastructure, which lives in file `parse.c`, is described below.

S194a. ⟨*tableparsing.c* S194a⟩≡ S197b ▷
  ⟨*private function prototypes for parsing* S199d⟩

G.2 Components, reduce functions, and form codes

A parser consumes *inputs* and puts *components* into an array. (Inputs are `Par`s and components are abstract syntax.) A reduce function takes the components in the array and reduces the them to a single node: an even bigger abstract-syntax tree, which may then be stored as a component in another array. "Reduction" is done by applying the build function for the node to the components that are reduced. In Impcore, a component is an expression, a list of expressions, a name, or a list of names.

S194b. ⟨*structure definitions for Impcore* S194b⟩≡ (S295a)

```
struct Component {
    Exp exp;
    Explist exps;
    Name name;
    Namelist names;
};
```

If you're a seasoned C programmer, you might think that the "right" representation of the component abstraction is a `union`, not a `struct`. But unions are unsafe. By using a struct, I give myself a fighting chance to debug the code. If I make a mistake and pick the wrong component, a memory-checking tool like Valgrind (Section 4.10.2, page 291) will detect the error.

The standard reduce functions are `reduce_to_exp` and `reduce_to_xdef`. They take similar arguments: the first argument codes for what kind of node the components should be reduced to, and the second argument points to an *array* that holds the components.

S195a. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁S192b S197c▷

```
Exp  reduce_to_exp (int alt, struct Component *components);
XDef reduce_to_xdef(int alt, struct Component *components);
```

As an example, here's the reduce function for Impcore expressions:

S195b. ⟨*parse.c* S195b⟩≡ S196a▷

```
Exp reduce_to_exp(int code, struct Component *components) {
    switch(code) {
    case SET:    return mkSet   (components[0].name, components[1].exp);
    case IFX:    return mkIfx   (components[0].exp, components[1].exp,
                                 components[2].exp);
    case WHILEX: return mkWhilex(components[0].exp, components[1].exp);
    case BEGIN:  return mkBegin (components[0].exps);
    case APPLY:  return mkApply (components[0].name, components[1].exps);
    ⟨cases for Impcore's reduce_to_exp added in exercises S195c⟩
    default:     assert(0);  // incorrectly configured parser
    }
}
```

To extend this function, just add more cases in the spot marked ⟨*cases for Impcore's* `reduce_to_exp` *added in exercises* S195c⟩.

S195c. ⟨*cases for Impcore's* `reduce_to_exp` *added in exercises* S195c⟩≡ (S195b) S209b▷

```
/* add your syntactic extensions here */
```

The trickiest part of writing a reduce function is figuring out the integer codes. Codes for expressions are easy: all expressions are represented by abstract syntax of the same C type, so I already have the perfect codes—the C enumeration literals used in the `alt` field of an `Exp`. Codes for extended definitions are more complicated: sometimes an extended definition is an `XDef` directly, but more often it is a `Def` or a `UnitTest`. And unfortunately, the `alt` fields for all three forms overlap. For example, code 1 means `EXP` as a `Def`, `CHECK_ERROR` as a `UnitTest`, and `USE` as an `XDef`. All three of these forms are ultimately extended definitions, so to distinguish among them, I need a more elaborate coding scheme. Here it is:

Code Range	In C	Meaning
0–99	`ANEXP(`*alt*`)`	Expressions
100–199	`ADEF(`*alt*`)`	Definitions
200–299	`ATEST(`*alt*`)`	Unit tests
300–399	`ANXDEF(`*alt*`)`	Other extended definitions
400–499	`ALET(`*alt*`)`	`LET` expressions used in Chapter 2
500–599	`SUGAR(`*alt*`)`	Syntactic sugar
1000	`LATER`	Syntax used in a later chapter
1001	`EXERCISE`	Syntax to be added for an Exercise

`type Exp`	𝒜
`type Explist`	S292d
`mkApply`	𝒜
`mkBegin`	𝒜
`mkIfx`	𝒜
`mkSet`	𝒜
`mkWhilex`	𝒜
`type Name`	43a
`type Namelist`	43a
`type XDef`	𝒜

In the table, *alt* stands for an enumeration literal of the sort to go in an `alt` field.

So they can appear after `case` in `switch`, codes are defined using C macros:

S195d. ⟨*macro definitions used in parsing* S195d⟩≡ (S295a)

```
#define ANEXP(ALT)  (  0+(ALT))
#define ADEF(ALT)   (100+(ALT))
#define ATEST(ALT)  (200+(ALT))
#define ANXDEF(ALT) (300+(ALT))
#define ALET(ALT)   (400+(ALT))
#define SUGAR(CODE) (500+(CODE))
#define LATER       1000
#define EXERCISE    1001
```

With the codes in place, I can write the reduce function for extended definitions.

S196a. ⟨*parse.c* S195b⟩+≡ ◁S195b S202f▷

```
XDef reduce_to_xdef(int alt, struct Component *comps) {
    switch(alt) {
    case ADEF(VAL):    return mkDef(mkVal(comps[0].name, comps[1].exp));
    case ADEF(DEFINE): return mkDef(mkDefine(comps[0].name,
                                              mkUserfun(comps[1].names, comps[2].exp)));
    case ANXDEF(USE):  return mkUse(comps[0].name);
    case ATEST(CHECK_EXPECT):
                       return mkTest(mkCheckExpect(comps[0].exp, comps[1].exp));
    case ATEST(CHECK_ASSERT):
                       return mkTest(mkCheckAssert(comps[0].exp));
    case ATEST(CHECK_ERROR):
                       return mkTest(mkCheckError(comps[0].exp));
    case ADEF(EXP):    return mkDef(mkExp(comps[0].exp));
    default:           assert(0);  // incorrectly configured parser
                       return NULL;
    }
}
```

G.3 Parser state and shift functions

A table-driven parser converts an input `Parlist` into components. There are at most `MAXCOMPS` components. (The value of `MAXCOMPS` must be at least the number of children that can appear in any node of any abstract-syntax tree. To support Exercise 30 on page 86, which has four components in the `define` form, I set `MAXCOMPS` to 4.) Inputs and components both go into a data structure. And if no programmer ever made a mistake, inputs and components would be enough. But because programmers do make mistakes, the data structure includes additional context, which can be added to an error message. The context I use includes the concrete syntax being parsed, the location where it came from, and the keyword or function name involved, if any.

S196b. ⟨*shared structure definitions* S196b⟩≡ (S295a) S201a▷

```
#define MAXCOMPS 4 // max # of components in any syntactic form
struct ParserState {
    int nparsed;       // number of components parsed so far
    struct Component components[MAXCOMPS];  // those components
    Parlist input;     // the part of the input not yet parsed

    struct ParsingContext {   // context of this parse
        Par par;       // the original thing being parsed
        struct Sourceloc {
            int line;                // current line number
            const char *sourcename;  // where the line came from
        } *source;
        Name name;     // a keyword, or name of a function being defined
    } context;
};
```

The important invariant of this data structure is that `components`$[i]$ is meaningful if and only if $0 \leq i <$ `nparsed`.

I define type abbreviations for `ParserState` and `ParsingContext`.

S197a. ⟨*shared type definitions* S197a⟩≡ (S295a) S197d ▷

```
typedef struct ParserState *ParserState;
typedef struct ParsingContext *ParsingContext;
```

A new parser state is created from a `Par` to be parsed, as well as the source of the `Par`. Those parameters provide the states' input and part of its context. The state's output is empty.

S197b. ⟨*tableparsing.c* S194a⟩+≡ ◁ S194a S198a ▷

```
struct ParserState mkParserState(Par p, Sourceloc source) {
    assert(p->alt == LIST);
    assert(source != NULL && source->sourcename != NULL);
    struct ParserState s;
    s.input          = p->list;
    s.context.par    = p;
    s.context.source = source;
    s.context.name   = NULL;
    s.nparsed        = 0;
    return s;
}
```

S197c. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁ S195a S197f ▷

```
struct ParserState mkParserState(Par p, Sourceloc source);
```

Each form of component is parsed by its own *shift function*. Why "shift"? Think of the `ParserState` as the state of a machine that puts components on the left and the input on the right. A shift function removes initial inputs and appends to components; this action "shifts" information from right to left. Shifting plays a role in several varieties of parsing technology.

A shift function normally updates the inputs and components in the parser state. A shift function also returns one of these results:

S197d. ⟨*shared type definitions* S197a⟩+≡ (S295a) ◁ S197a S197e ▷

```
typedef enum ParserResult {
  PARSED,            // some input was parsed without any errors
  INPUT_EXHAUSTED,   // there aren't enough inputs
  INPUT_LEFTOVER,    // there are too many inputs
  BAD_INPUT,         // an input wasn't what it should have been
  STOP_PARSING       // all the inputs have been parsed; it's time to stop
} ParserResult;
```

When a shift function runs out of input or sees input left over, it returns either `INPUT_EXHAUSTED` or `INPUT_LEFTOVER`. Returning one of these error results is better than simply calling `synerror`, because the calling function knows what row it's trying to parse and so can issue a better error message. But for other error conditions, shift functions can call `synerror` directly.

The C type of a shift function is `ShiftFun`.

S197e. ⟨*shared type definitions* S197a⟩+≡ (S295a) ◁ S197d S201d ▷

```
typedef ParserResult (*ShiftFun)(ParserState);
```

The four basic shift functions shift expressions and names:

S197f. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁ S197c S198b ▷

```
ParserResult sExp     (ParserState state);  // shift 1 input into Exp
ParserResult sExps    (ParserState state);  // shift all inputs into Explist
ParserResult sName    (ParserState state);  // shift 1 input into Name
ParserResult sNamelist(ParserState state);  // shift 1 input into Namelist
```

These functions have short names because below I put them in arrays: *I represent a syntactic form's components as an array of shift functions.* This dirty trick is inspired by the functional-programming techniques described in Chapter 2. But before I put the shift functions in arrays, let's see their implementations.

A shift operation is divided into two halves. The first half removes an input and ensures that there is room for a component. The second half writes the component and updates `nparsed`. The first half is the same for every shift function, and it looks like this:

S198a. ⟨*tableparsing.c* S194a⟩+≡ ◁S197b S198c▷

```
void halfshift(ParserState s) {
    assert(s->input);
    s->input = s->input->tl;  // move to the next input
    assert(s->nparsed < MAXCOMPS);
}
```

S198b. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁S197f S198e▷

```
void halfshift(ParserState state); // advance input, check for room in output
```

A full shift calls `halfshift` and then places a component. The full shift for an expression is defined as function `sExp`. It calls `parseexp`, with which it is mutually recursive.

S198c. ⟨*tableparsing.c* S194a⟩+≡ ◁S198a S198d▷

```
ParserResult sExp(ParserState s) {
    if (s->input == NULL) {
        return INPUT_EXHAUSTED;
    } else {
        Par p = s->input->hd;
        halfshift(s);
        s->components[s->nparsed++].exp = parseexp(p, s->context.source);
        return PARSED;
    }
}
```

Function `sExps` converts the *entire* input into an `Explist`. The `halfshift` isn't useful here. And a `NULL` input is OK; it just parses into an empty `Explist`.

S198d. ⟨*tableparsing.c* S194a⟩+≡ ◁S198c S199a▷

```
ParserResult sExps(ParserState s) {
    Explist es = parseexplist(s->input, s->context.source);
    assert(s->nparsed < MAXCOMPS);
    s->input = NULL;
    s->components[s->nparsed++].exps = es;
    return PARSED;
}
```

Function `parseexplist` is defined below with the other parsing functions.

S198e. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁S198b S199b▷

```
Explist parseexplist(Parlist p, Sourceloc source);
```

Function `sName` is structured just like `sExp`; the only difference is that where `sExp` calls `parseexp`, `sName` calls `parsename`.

S199a. ⟨*tableparsing.c* S194a⟩+≡ ◁S198d S199c▷

```
ParserResult sName(ParserState s) {
    if (s->input == NULL) {
        return INPUT_EXHAUSTED;
    } else {
        Par p = s->input->hd;
        halfshift(s);
        s->components[s->nparsed++].name = parsename(p, &s->context);
        return PARSED;
    }
}
```

Notice that `parsename`, which is defined below, takes the current context as an extra parameter. That context enables `parsename` to give a good error message if it encounters an input that is *not* a valid name.

S199b. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁S198e S200a▷

```
Name parsename(Par p, ParsingContext context);
```

A `Namelist` appears in parentheses and is used only in the `define` form.

S199c. ⟨*tableparsing.c* S194a⟩+≡ ◁S199a S199e▷

```
ParserResult sNamelist(ParserState s) {
    if (s->input == NULL) {
        return INPUT_EXHAUSTED;
    } else {
        Par p = s->input->hd;
        switch (p->alt) {
        case ATOM:
            synerror(s->context.source,
                     "%p: usage: (define fun (formals) body)",
                     s->context.par);
        case LIST:
            halfshift(s);
            s->components[s->nparsed++].names =
                parsenamelist(p->list, &s->context);
            return PARSED;
        }
        assert(0);
    }
}
```

S199d. ⟨*private function prototypes for parsing* S199d⟩≡ (S194a) S202c▷

```
static Namelist parsenamelist(Parlist ps, ParsingContext context);
```

These shift functions aren't used just to move information from input to components. A *sequence* of shift functions represents the components that are expected to be part of a syntactic form. (This technique of using functions as data is developed at length in Chapter 2.) The syntactic form is parsed by calling its functions in sequence. The end of the sequence is marked by function `stop`. This function checks to be sure all input is consumed and signals that it is time to stop parsing. Unlike the other shift functions, it does not change the `state`.

S199e. ⟨*tableparsing.c* S194a⟩+≡ ◁S199c S200b▷

```
ParserResult stop(ParserState state) {
    if (state->input == NULL)
        return STOP_PARSING;
    else
        return INPUT_LEFTOVER;
}
```

S200a. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁S199b S200c▷

```
ParserResult stop(ParserState state);
```

Finally, I define a special shift function that doesn't do any shifting. Instead, it sets the context for parsing a function definition. In the list of parsing functions for a function definition, the `sName` that parses the function's name is followed immediately by `setcontextname`.

S200b. ⟨*tableparsing.c* S194a⟩+≡ ◁S199e S200d▷

```
ParserResult setcontextname(ParserState s) {
    assert(s->nparsed > 0);
    s->context.name = s->components[s->nparsed-1].name;
    return PARSED;
}
```

S200c. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁S200a S200f▷

```
ParserResult setcontextname(ParserState state);
```

I define one more shift function, which is meant to help you with Exercise 30 in Chapter 1. This exercise asks you to add local variables to Impcore. A declaration of local variables, if present, is parsed with shift function `sLocals`. This function looks for the keyword `locals`. If found, the keyword marks a list of the names of local variables, and this list of names is shifted into the `s->components` array. If the keyword `locals` is not found, there are no local variables, and a `NULL` pointer is shifted into the `s->components` array.

S200d. ⟨*tableparsing.c* S194a⟩+≡ ◁S200b S201b▷

```
ParserResult sLocals(ParserState s) {
    Par p = s->input ? s->input->hd : NULL;  // useful abbreviation
    if (⟨Par p represents a list beginning with keyword locals S200e⟩) {
        struct ParsingContext context;
        context.name = strtoname("locals");
        context.par = p;
        halfshift(s);
        s->components[s->nparsed++].names = parsenamelist(p->list->tl, &context);
        return PARSED;
    } else {
        s->components[s->nparsed++].names = NULL;
        return PARSED;
    }
}
```

The keyword test is just complicated enough that it warrants being put in a named code chunk.

S200e. ⟨`Par p` *represents a list beginning with keyword* `locals` S200e⟩≡ (S200d)

```
p != NULL && p->alt == LIST && p->list != NULL &&
p->list->hd->alt == ATOM && p->list->hd->atom == strtoname("locals")
```

S200f. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁S200c S201c▷

```
ParserResult sLocals(ParserState state);
                  // shift locals if [locals x y z ...]
```

G.4 Representing and parsing tables and rows

The shift functions defined above go into rows. As shown in Tables G.1 and G.2 on page S193, a row needs a keyword, a code, and a sequence of components. The sequence of components is represented as an array of shift functions ending in `stop`.

S201a. ⟨*shared structure definitions* S196b⟩+≡ (S295a) ◁ S196b

```
struct ParserRow {
    const char *keyword;
    int code;
    ShiftFun *shifts;  // points to array of shift functions
};
```

To parse an input using a row, function `rowparse` calls shift functions until a shift function says to stop—or detects an error.

S201b. ⟨*tableparsing.c* S194a⟩+≡ ◁ S200d S201e ▷

```
void rowparse(struct ParserRow *row, ParserState s) {
    ShiftFun *f = &row->shifts[0];

    for (;;) {
        ParserResult r = (*f)(s);
        switch (r) {
        case PARSED:          f++; break;
        case STOP_PARSING:    return;
        case INPUT_EXHAUSTED:
        case INPUT_LEFTOVER:
        case BAD_INPUT:       usage_error(row->code, r, &s->context);
        }
    }
}
```

S201c. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁ S200f S202a ▷

```
void rowparse(struct ParserRow *table, ParserState s);
void usage_error(int alt, ParserResult r, ParsingContext context);
```

The `usage_error` function is discussed below. Meanwhile, `rowparse` is called by `tableparse`, which looks for a keyword in the input, and if it finds one, uses the matching row to parse. Otherwise, it uses the final row, which it identifies by the `NULL` keyword.

S201d. ⟨*shared type definitions* S197a⟩+≡ (S295a) ◁ S197e S209a ▷

```
typedef struct ParserRow *ParserTable;
```

S201e. ⟨*tableparsing.c* S194a⟩+≡ ◁ S201b S202b ▷

```
struct ParserRow *tableparse(ParserState s, ParserTable t) {
    if (s->input == NULL)
        synerror(s->context.source, "%p: unquoted empty parentheses",
                                    s->context.par);

    Name first = s->input->hd->alt == ATOM ? s->input->hd->atom : NULL;
                     // first Par in s->input, if it is present and an atom

    unsigned i;  // to become the index of the matching row in ParserTable t
    for (i = 0; !rowmatches(&t[i], first); i++)
        ;
    ⟨adjust the state s so it’s ready to start parsing using row t[i] S202d⟩
    rowparse(&t[i], s);
    return &t[i];
}
```

`halfshift`	S198b
`type Name`	43a
`type Par`	$\mathcal{A}$
`parsenamelist`	S199d
`type ParserResult`	S197d
`type ParserState`	S197a
`type Parsing-Context`	S197a
`rowmatches`	S202c
`type ShiftFun`	S197e
`strtoname`	43b
`synerror`	47b

S202a. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁ S201c S205b ▷

```
struct ParserRow *tableparse(ParserState state, ParserTable t);
```

A row matches if the row's keyword is `NULL` or if the keyword stands for the same name as `first`.

S202b. ⟨*tableparsing.c* S194a⟩+≡ ◁ S201e S203a ▷

```
static bool rowmatches(struct ParserRow *row, Name first) {
    return row->keyword == NULL || strtoname(row->keyword) == first;
}
```

S202c. ⟨*private function prototypes for parsing* S199d⟩+≡ (S194a) ◁ S199d S208b ▷

```
static bool rowmatches(struct ParserRow *row, Name first);
```

Once a row has matched, the parser state might have to be adjusted. If row `t[i]` has a keyword, then the first `Par` in the input is that keyword, and that `Par` needs to be consumed—so function `tableparse` adjusts `s->input` and sets the context.

S202d. ⟨*adjust the state* `s` *so it's ready to start parsing using row* `t[i]` S202d⟩≡ (S201e)

```
if (t[i].keyword) {
    assert(first != NULL);
    s->input = s->input->tl;
    s->context.name = first;
}
```

G.5 Parsing tables and functions

Every language has two parsing tables: one for expressions and one for extended definitions.

S202e. ⟨*declarations of globals used in lexical analysis and parsing* S202e⟩≡ (S295a) S206b ▷

```
extern struct ParserRow exptable[];
extern struct ParserRow xdeftable[];
```

Here, as promised from Table G.1 (page S193), is `exptable`: the parsing table for Impcore expressions. Each row of `exptable` refers to an array of shift functions, which must be defined separately and given its own name.

S202f. ⟨*parse.c* S195b⟩+≡ ◁ S196a S203b ▷

```
static ShiftFun setshifts[]   = { sName, sExp,          stop };
static ShiftFun ifshifts[]    = { sExp,  sExp, sExp,    stop };
static ShiftFun whileshifts[] = { sExp,  sExp,          stop };
static ShiftFun beginshifts[] = { sExps,                stop };
static ShiftFun applyshifts[] = { sName, sExps,         stop };

⟨arrays of shift functions added to Impcore in exercises S203c⟩

struct ParserRow exptable[] = {
  { "set",   SET,    setshifts },
  { "if",    IFX,    ifshifts },
  { "while", WHILEX, whileshifts },
  { "begin", BEGIN,  beginshifts },
  ⟨rows added to Impcore's exptable in exercises S203d⟩
  { NULL,    APPLY,  applyshifts }  // must come last
};
```

The table exptable is used by parsing function parseexp. Function parseexp delegates the heavy lifting to other functions: given an atom, it calls exp_of_atom, and given a list, it calls tableparse and reduce_to_exp.

S203a. ⟨*tableparsing.c* S194a⟩+≡ ◁S202b S204a▷

```
Exp parseexp(Par p, Sourceloc source) {
    switch (p->alt) {
    case ATOM:
        ⟨if p->atom is a reserved word, call synerror with source S206a⟩
        return exp_of_atom(source, p->atom);
    case LIST:
        {   struct ParserState s = mkParserState(p, source);
            struct ParserRow *row = tableparse(&s, exptable);
            if (row->code == EXERCISE) {
                synerror(source, "implementation of %n is left as an exercise",
                         s.context.name);
            } else {
                Exp e = reduce_to_exp(row->code, s.components);
                check_exp_duplicates(source, e);
                return e;
            }
        }
    }
    assert(0);
}
```

In later chapters, function parseexp is reused with different versions of functions exp_of_atom, exptable, and reduce_to_exp.

In Impcore, exp_of_atom classifies each atom as either an integer literal or a variable.

S203b. ⟨*parse.c* S195b⟩+≡ ◁S202f S206c▷

```
Exp exp_of_atom(Sourceloc loc, Name atom) {
    const char *s = nametostr(atom);
    char *t;   // to point to the first non-digit in s
    long l = strtol(s, &t, 10);
    if (*t != '\0') // the number is just a prefix
        return mkVar(atom);
    else if (((l == LONG_MAX || l == LONG_MIN) && errno == ERANGE) ||
             l > (long)INT32_MAX || l < (long)INT32_MIN)
    {
        synerror(loc, "arithmetic overflow in integer literal %s", s);
        return NULL; // unreachable
    } else {  // the number is the whole atom, and not too big
        return mkLiteral(l);
    }
}
```

More syntax can be added in exercises.

S203c. ⟨*arrays of shift functions added to Impcore in exercises* S203c⟩≡ (S202f) S210a▷

```
/* for each new row added to exptable, add an array of shift functions here */
```

S203d. ⟨*rows added to Impcore's* exptable *in exercises* S203d⟩≡ (S202f) S210b▷

```
/* add a row here for each new syntactic form of Exp */
```

check_exp_duplicates	S208d
type Exp	𝒜
exp_of_atom	S192b
first	S201e
mkLiteral	𝒜
mkParserState	S197c
mkVar	𝒜
type Name	43a
nametostr	43b
type Par	𝒜
type ParserState	S197a
type ParserTable	S201d
reduce_to_exp	S195a
sExp	S197f
sExps	S197f
type ShiftFun	S197e
sName	S197f
type Sourceloc	S293h
stop	S200a
strtoname	43b
synerror	47b

Impcore's other parsing table and function handle extended definitions. The extended-definition table is shared among several languages. Because it is shared, I put it in `tableparsing.c`, not in `parse.c`.

S204a. ⟨*tableparsing.c* S194a⟩+≡ ◁ S203a S204b ▷

```
static ShiftFun valshifts[]      = { sName, sExp,           stop };
static ShiftFun defineshifts[]   = { sName, setcontextname,
                                     sNamelist, sExp,       stop };
static ShiftFun useshifts[]      = { sName,                 stop };
static ShiftFun checkexpshifts[] = { sExp, sExp,            stop };
static ShiftFun checkassshifts[] = { sExp,                  stop };
static ShiftFun checkerrshifts[] = { sExp,                  stop };
static ShiftFun expshifts[]      = { use_exp_parser };

void extendDefine(void) { defineshifts[3] = sExps; }

struct ParserRow xdeftable[] = {
    { "val",          ADEF(VAL),           valshifts },
    { "define",       ADEF(DEFINE),        defineshifts },
    { "use",          ANXDEF(USE),         useshifts },
    { "check-expect", ATEST(CHECK_EXPECT), checkexpshifts },
    { "check-assert", ATEST(CHECK_ASSERT), checkassshifts },
    { "check-error",  ATEST(CHECK_ERROR),  checkerrshifts },
    ⟨rows added to xdeftable in exercises S210d⟩
    { NULL,           ADEF(EXP),           expshifts }  // must come last
};
```

Function `parsexdef` is quite similar to `parseexp`.

S204b. ⟨*tableparsing.c* S194a⟩+≡ ◁ S204a S205a ▷

```
XDef parsexdef(Par p, Sourceloc source) {
    switch (p->alt) {
    case ATOM:
        return mkDef(mkExp(parseexp(p, source)));
    case LIST:;
        struct ParserState s  = mkParserState(p, source);
        struct ParserRow *row = tableparse(&s, xdeftable);
        XDef d = reduce_to_xdef(row->code, s.components);
        if (d->alt == DEF)
            check_def_duplicates(source, d->def);
        return d;
    }
    assert(0);
}
```

The case for a top-level `ADEF(EXP)` node has just one component, an `Exp`, which is parsed using the sequences of parsers `expshifts`. But by the time `tableparse` gets to `expshifts`, the input isn't a `Par` any more—instead, it's the *list* of `Par`s that appeared inside `LIST`. If `expshifts` tried to use `sExp`, it would fail to parse an expression like `(+ 1 2)`: function `tableparse` would unpack the list of atoms inside the brackets, `sExp` would try to parse `+` as an expression, and the `1` and `2` would be left over.

What's needed is to go back to the original `Par` that was input, and to pass it to `parseexp`. That's done by shift function `use_exp_parser`, which is the only "component" in `expshifts`.

S205a. ⟨*tableparsing.c* S194a⟩+≡ ◁S204b S205c▷

```
ParserResult use_exp_parser(ParserState s) {
    Exp e = parseexp(s->context.par, s->context.source);
    halfshift(s);
    s->components[s->nparsed++].exp = e;
    return STOP_PARSING;
}
```

S205b. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁S202a S208c▷

```
ParserResult use_exp_parser(ParserState state);
```

The next parsing function is `parsename`. This function could accept only names, and it would work on every syntactically correct program. But programmers make mistakes, and when a program is not syntactically correct, I'd like my interpreter to issue a helpful error message. In this case I can do it by allowing `parsename` to handle any *expression*, and if the expression *isn't* a name, `parsename` can issue an error message that is aware of the context. The message is actually issued by function `name_error`, which is defined below.

S205c. ⟨*tableparsing.c* S194a⟩+≡ ◁S205a S205d▷

```
Name parsename(Par p, ParsingContext context) {
    Exp e = parseexp(p, context->source);
    if (e->alt != VAR)
        return name_error(p, context);
    else
        return e->var;
}
```

In addition to a parser for expressions and a parser for names, Impcore needs a parser for a list of expressions and a parser for a list of names. A list of expressions is parsed recursively.

S205d. ⟨*tableparsing.c* S194a⟩+≡ ◁S205c S205e▷

```
Explist parseexplist(Parlist input, Sourceloc source) {
    if (input == NULL) {
        return NULL;
    } else {
        Exp     e  = parseexp    (input->hd, source);
        Explist es = parseexplist(input->tl, source);
        return mkEL(e, es);
    }
}
```

A list of names is also parsed recursively.

S205e. ⟨*tableparsing.c* S194a⟩+≡ ◁S205d S207a▷

```
static Namelist parsenamelist(Parlist ps, ParsingContext context) {
    if (ps == NULL) {
        return NULL;
    } else {
        Exp e = parseexp(ps->hd, context->source);
        if (e->alt != VAR)
            synerror(context->source,
                     "in %p, formal parameters of %n must be names, "
                     "but %p is not a name",
                     context->par, context->name, ps->hd);
        return mkNL(e->var, parsenamelist(ps->tl, context));
    }
}
```

G.6 Error detection and handling

My code handles four classes of errors: misuse of a reserved word like `if` or `while`, wrong number of components, failure to deliver a name when a name is expected, and a duplicate name where distinct names are expected.

Misuse of reserved words is detected by the following check, which prevents such oddities as a user-defined function named `if`. A word is reserved if it appears in `exptable` or `xdeftable`.

S206a. ⟨*if* `p->atom` *is a reserved word, call* `synerror` *with* `source` S206a⟩≡ (S203a)

```
for (struct ParserRow *entry = exptable; entry->keyword != NULL; entry++)
    if (p->atom == strtoname(entry->keyword))
        synerror(source, "%n is a reserved word and may not be used "
                 "to name a variable or function", p->atom);
for (struct ParserRow *entry = xdeftable; entry->keyword != NULL; entry++)
    if (p->atom == strtoname(entry->keyword))
        synerror(source, "%n is a reserved word and may not be used "
                 "to name a variable or function", p->atom);
```

When a parser sees an input that has the wrong number of components, as in `(if p (set x 5))` or `(set x y z)`, it calls `usage_error` with a code, a `ParserResult`, and a context. The code is looked up in `usage_table`, which contains a sample string showing what sort of syntax was expected.

S206b. ⟨*declarations of globals used in lexical analysis and parsing* S202e⟩+≡ (S295a) ◁S202e

```
extern struct Usage {
    int code;             // codes for form in reduce_to_exp or reduce_to_xdef
    const char *expected; // shows the expected usage of the identified form
} usage_table[];
```

S206c. ⟨*parse.c* S195b⟩+≡ ◁S203b S208e▷

```
struct Usage usage_table[] = {
    { ADEF(VAL),           "(val x e)" },
    { ADEF(DEFINE),        "(define fun (formals) body)" },
    { ANXDEF(USE),         "(use filename)" },
    { ATEST(CHECK_EXPECT), "(check-expect exp-to-run exp-expected)" },
    { ATEST(CHECK_ASSERT), "(check-assert exp)" },
    { ATEST(CHECK_ERROR),  "(check-error exp)" },
    { SET,                 "(set x e)" },
    { IFX,                 "(if cond true false)" },
    { WHILEX,              "(while cond body)" },
    { BEGIN,               "(begin exp ... exp)" },
    ⟨Impcore usage_table entries added in exercises S210c⟩
    { -1, NULL }  // marks end of table
};
```

Strictly speaking, if you add new syntax to a language, you should extend not only the parsing table and the reduce function, but also the `usage_table`. If there is no usage string for a given code, function `usage_error` can't say what the expected usage is.

Function usage_error searches usage_table, then spits out an error message.

S207a. ⟨*tableparsing.c* S194a⟩+≡ ◁ S205e S207b ▷

```
void usage_error(int code, ParserResult why_bad, ParsingContext context) {
    for (struct Usage *u = usage_table; u->expected != NULL; u++)
        if (code == u->code) {
            const char *message;
            switch (why_bad) {
            case INPUT_EXHAUSTED:
                message = "too few components in %p; expected %s";
                break;
            case INPUT_LEFTOVER:
                message = "too many components in %p; expected %s";
                break;
            default:
                message = "badly formed input %p; expected %s";
                break;
            }
            synerror(context->source, message, context->par, u->expected);
        }
    synerror(context->source,
             "something went wrong parsing %p", context->par);
}
```

Finally, if parsename sees something other than a name, it calls name_error. The error message says what went wrong and what the context is. The context is identified by c->name, and to make extending name_error as easy as possible, I first convert c->name to an integer code. The matching code can be scrutinized using a switch statement.

S207b. ⟨*tableparsing.c* S194a⟩+≡ ◁ S207a S208a ▷

```
void *name_error(Par bad, struct ParsingContext *c) {
    switch (code_of_name(c->name)) {
    case ADEF(VAL):
        synerror(c->source, "in %p, expected (val x e), but %p is not a name",
                 c->par, bad);
    case ADEF(DEFINE):
        synerror(c->source, "in %p, expected (define f (x ...) e), "
                            "but %p is not a name",
                 c->par, bad);
    case ANXDEF(USE):
        synerror(c->source, "in %p, expected (use filename), "
                            "but %p is not a filename",
                 c->par, bad);
    case SET:
        synerror(c->source, "in %p, expected (set x e), but %p is not a name",
                 c->par, bad);
    case APPLY:
        synerror(c->source, "in %p, expected (function-name ...), "
                            "but %p is not a name",
                 c->par, bad);
    default:
        synerror(c->source, "in %p, expected a name, but %p is not a name",
                 c->par, bad);
    }
}
```

code_of_name	S208c
exptable	S202e
type Par	𝒜
type ParserResult	S197d
type ParsingContext	S197a
source	S203a
strtoname	43b
synerror	47b
xdeftable	S202e

The code is produced by function `code_of_name`, which does a reverse lookup in `exptable` and `xdeftable`.

S208a. ⟨*tableparsing.c* S194a⟩+≡ ◁ S207b

```
int code_of_name(Name n) {
    struct ParserRow *entry;
    for (entry = exptable; entry->keyword != NULL; entry++)
        if (n == strtoname(entry->keyword))
            return entry->code;
    if (n == NULL)
        return entry->code;
    for (entry = xdeftable; entry->keyword != NULL; entry++)
        if (n == strtoname(entry->keyword))
            return entry->code;
    assert(0);
}
```

S208b. ⟨*private function prototypes for parsing* S199d⟩+≡ (S194a) ◁ S202c

```
void *name_error(Par bad, struct ParsingContext *context);
                     // expected a name, but got something else
```

S208c. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁ S205b S208d ▷

```
int code_of_name(Name n);
```

In addition to syntax errors, parsers can also detect duplicate names. In general, duplicate names can occur in an expression or in an extended definition.

S208d. ⟨*shared function prototypes* S192a⟩+≡ (S295a) ◁ S208c

```
void check_exp_duplicates(Sourceloc source, Exp e);
void check_def_duplicates(Sourceloc source, Def d);
```

In Impcore, a rule of operational semantics requires that in every function definition, the names of the formal parameters be distinct:

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{DEFINE}(f, \langle x_1, \ldots, x_n \rangle, e), \xi, \phi \rangle \rightarrow \langle \xi, \phi\{f \mapsto \text{USER}(\langle x_1, \ldots, x_n \rangle, e)\} \rangle}. \qquad (\textsc{DefineFunction})$$

If the rule is violated, `check_def_duplicates` reports the violation along with a source-code location.

S208e. ⟨*parse.c* S195b⟩+≡ ◁ S206c S208f ▷

```
void check_def_duplicates(Sourceloc source, Def d) {
    if (d->alt == DEFINE && duplicatename(d->define.userfun.formals) != NULL)
        synerror(source,
                 "Formal parameter %n appears twice "
                 "in definition of function %n",
                 duplicatename(d->define.userfun.formals), d->define.name);
}
```

Impcore has no expressions that bind names, so the expression check does nothing.

S208f. ⟨*parse.c* S195b⟩+≡ ◁ S208e

```
void check_exp_duplicates(Sourceloc source, Exp e) {
    (void)source; (void)e;
}
```

Here are integer codes for all the syntactic forms that are suggested to be implemented as syntactic sugar.

S209a. ⟨*shared type definitions* S197a⟩+≡ (S295a) ◁S201d

```
enum Sugar {
  CAND, COR,    // short-circuit Boolean operators

  WHILESTAR, DO_WHILE, FOR,     // bonus loop forms

  WHEN, UNLESS,       // single-sided conditionals

  RECORD,             // record-type definition

  COND                // McCarthy's conditional from Lisp

};
```

Figure G.3: Codes used for syntactic sugar in Chapters 1 to 3

G.7 Extending Impcore with syntactic sugar

Design for extension is all very well, but examples are even better. In this section I extend Impcore short-circuit `&&` and `||` operators, like those found in C. Unlike Impcore's functions `and` and `or`, C's syntactic forms `&&` and `||` don't always evaluate all their operands. For example, in code chunk ⟨`Par p` *represents a list beginning with keyword* `locals` S200e⟩, it is absolutely critical that `p->alt` be evaluated only when `p` is not `NULL`. (Dereferencing a null pointer typically causes a fault that crashes the program.) In Impcore, these operators can be defined by syntactic sugar:

$$\texttt{(\&\&}\ e_1\ e_2\texttt{)} \overset{\triangle}{=} \texttt{(if}\ e_1\ e_2\ \texttt{0)}$$
$$\texttt{(||}\ e_1\ e_2\texttt{)} \overset{\triangle}{=} \texttt{(if}\ e_1\ \texttt{1}\ e_2\ \texttt{)}$$

Operator `&&` evaluates e_2 only if e_1 is nonzero; dually, `||` evaluates e_2 only if e_1 is zero. These versions behave differently from the basis functions `and` and `or`, which always evaluate both arguments.

For `&&` and `||`, as for any other new expression, I have to add five things:

1. Integer codes for the new expressions
2. New cases for the `reduce_to_exp` function
3. New arrays of shift functions (unless an existing array can be reused)
4. New rows for `exptable`
5. New rows for `usage_table`

The most interesting of these is the reduce function, which expands the new form into existing syntax. The new codes are named `CAND` and `COR`, which stand for "conditional *and*" and "conditional *or*"; these names were used in the programming language Algol W and in Dijkstra's (1976) unnamed language of "guarded commands."

S209b. ⟨*cases for Impcore's* `reduce_to_exp` *added in exercises* S195c⟩+≡ (S195b) ◁S195c

```
case SUGAR(CAND):
    return mkIfx(components[0].exp, components[1].exp, mkLiteral(0));
case SUGAR(COR):
    return mkIfx(components[0].exp, mkLiteral(1), components[1].exp);
```

`components`	S195b
`type Def`	$\mathcal{A}$
`duplicatename`	S184d
`type Exp`	$\mathcal{A}$
`exptable`	S202e
`mkIfx`	$\mathcal{A}$
`mkLiteral`	$\mathcal{A}$
`type Name`	43a
`type Par`	$\mathcal{A}$
`type Sourceloc`	S293h
`strtoname`	43b
`synerror`	47b
`xdeftable`	S202e

The components of a short-circuit conditional are the two subexpressions e_1 and e_2, so the extension needs an array of shift functions that shifts two expressions and then stops.

S210a. ⟨*arrays of shift functions added to Impcore in exercises* S203c⟩+≡ (S202f) ◁S203c

```
static ShiftFun conditionalshifts[] = { sExp, sExp, stop };
```

The `exptable` rows use the given shift functions, and the `usage_table` entries show the expected syntax.

S210b. ⟨*rows added to Impcore's* `exptable` *in exercises* S203d⟩+≡ (S202f) ◁S203d

```
{ "&&", SUGAR(CAND), conditionalshifts },
{ "||", SUGAR(COR),  conditionalshifts },
```

S210c. ⟨*Impcore* `usage_table` *entries added in exercises* S210c⟩≡ (S206c)

```
{ SUGAR(CAND), "(&& exp exp)" },
{ SUGAR(COR),  "(|| exp exp)" },
```

The conditional sugar doesn't require any new definition forms.

S210d. ⟨*rows added to* `xdeftable` *in exercises* S210d⟩≡ (S204a)

```
/* add new forms for extended definitions here */
```

Finally, here is a short demonstration showing how `&&` and `||` differ from `and` and `or`:

S210e. ⟨*transcript* S210e⟩≡

```
-> (|| 1 (println 99))
1
-> (or 1 (println 99))
99
1
-> (&& 0 (println 33))
0
-> (|| 0 (println 33))
33
33
```

Appendix H contents

Code for writing interpreters in ML H

Just as Appendix F presents reusable infrastructure for building interpreters in C, this appendix presents reusable infrastructure for building interpreters in ML. This infrastructure is shared among many interpreters, but the abstractions and implementations presented here are not as closely connected to the study of programming languages as the ones in the main text. (The shared infrastructure that is closely connected is presented in Chapter 5.)

Each interpreter that is written in ML incorporates all the following code chunks, some of which are defined in Chapter 5 and some of which are defined below.

S213a. ⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S213a⟩≡ (S379)

⟨*for working with curried functions:* `id`*,* `fst`*,* `snd`*,* `pair`*,* `curry`*, and* `curry3` S249b⟩
⟨*support for names and environments* S219d⟩
⟨*exceptions used in every interpreter* S213b⟩
⟨*support for detecting and signaling errors detected at run time* S219e⟩
⟨*list functions not provided by Standard ML's initial basis* S219a⟩
⟨*utility functions for string manipulation and printing* S214c⟩
⟨*support for representing errors as ML values* S221b⟩
⟨*type* `interactivity` *plus related functions and value* S236a⟩
⟨*simple implementations of set operations* S217b⟩
⟨*ability to interrogate environment variable* `BPCOPTIONS` S220a⟩
⟨*collections with mapping and combining functions* S218a⟩
⟨*suspensions* S228a⟩
⟨*streams* S228c⟩
⟨*stream transformers and their combinators* S246b⟩
⟨*support for source-code locations and located streams* S233b⟩
⟨*streams that track line boundaries* S258⟩
⟨*support for lexical analysis* S254b⟩
⟨*common parsing code* S246a⟩
⟨*shared utility functions for initializing interpreters* S240a⟩
⟨*function application with overflow checking* S220b⟩

All interpreters incorporate these two exceptions:

S213b. ⟨*exceptions used in every interpreter* S213b⟩≡ (S213a)

```
exception RuntimeError of string (* error message *)
exception LeftAsExercise of string (* string identifying code *)
```

All interpreters that include type checkers also incorporate these exceptions:

S213c. ⟨*exceptions used in languages with type checking* S213c⟩≡

```
exception TypeError of string
exception BugInTypeChecking of string
```

And all interpreters that implement type inference also incorporate these exceptions:

S213d. ⟨*exceptions used in languages with type inference* S213d⟩≡

```
exception TypeError of string
exception BugInTypeInference of string
```

H.1 Common syntactic forms

Syntactic forms for unit tests and for extended definitions are often shared.

The following forms of unit test are used by both of the major untyped languages in this book: μScheme (Chapter 5) and μSmalltalk (Chapter 10).

S214a. ⟨*definition of* `unit_test` *for untyped languages (shared)* S214a⟩≡ (S380a)

```
datatype unit_test = CHECK_EXPECT of exp * exp
                   | CHECK_ASSERT of exp
                   | CHECK_ERROR  of exp
```

And these forms of extended definition are used by all languages in Chapters 5 to 10.

S214b. ⟨*definition of* `xdef` *(shared)* S214b⟩≡ (S380a)

```
datatype xdef = DEF    of def
              | USE    of name
              | TEST   of unit_test
```

H.2 Reusable utility functions

The interpreters share some small utility functions for printing, for manipulating automatically generated names, and for manipulating sets.

H.2.1 Utility functions for creating and hashing strings

Standard ML's built-in support for converting integers to strings uses the ~ character as a minus sign. But the bridge languages represent a minus sign as a hyphen.

S214c. ⟨*utility functions for string manipulation and printing* S214c⟩≡ (S213a) S214d ▷

```
intString : int -> string
```

```
fun intString n =
  String.map (fn #"~" => #"-" | c => c) (Int.toString n)
```

To characterize a list by its length and contents, interpreter messages use strings like "3 arguments," which come from functions `plural` and `countString`.

S214d. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁ S214c S214e ▷

```
fun plural what [x] = what
  | plural what _   = what ^ "s"

fun countString xs what =
  intString (length xs) ^ " " ^ plural what xs
```

To separate items by spaces or commas, interpreters use `spaceSep` and `commaSep`, which are special cases of the basis-library function `String.concatWith`.

S214e. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁ S214d S214f ▷

```
spaceSep : string list -> string
commaSep : string list -> string
```

```
val spaceSep = String.concatWith " "   (* list separated by spaces *)
val commaSep = String.concatWith ", "  (* list separated by commas *)
```

Sometimes, as when printing substitutions for example, the empty list should be represented by something besides the empty string. Like maybe the string `"idsubst"`. Such output can be produced by, e.g., `nullOrCommaSep "idsubst"`.

S214f. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁ S214e S215a ▷

```
nullOrCommaSep : string -> string list -> string
```

```
fun nullOrCommaSep empty [] = empty
  | nullOrCommaSep _     ss = commaSep ss
```

The hash primitive in the μSmalltalk interpreter uses an algorithm by Glenn Fowler, Phong Vo, and Landon Curt Noll, which I implement in function `fnvHash`. I have adjusted the algorithm's "offset basis" by removing the high bit, so the computation works using 31-bit integers. The algorithm is described by an IETF draft at `http://tools.ietf.org/html/draft-eastlake-fnv-03`, and it's also described by the web page at `http://www.isthe.com/chongo/tech/comp/fnv/`.

S215a. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁S214f S215b▷

`fnvHash : string -> int`

```
fun fnvHash s =
  let val offset_basis = 0wx011C9DC5 : Word.word  (* trim the high bit *)
      val fnv_prime    = 0w16777619  : Word.word
      fun update (c, hash) = Word.xorb (hash, Word.fromInt (ord c)) * fnv_prime
      fun int w =
        Word.toIntX w handle Overflow => Word.toInt (Word.andb (w, 0wxffffff))
  in  int (foldl update offset_basis (explode s))
  end
```

H.2.2 Utility functions for printing

For writing values and other information to standard output, Standard ML provides a simple `print` primitive, which writes a string. Anything more sophisticated, such as writing to standard error, requires using the the `TextIO` module, which is roughly analogous to C's `<stdio.h>`. Using `TextIO` can be awkward, so I define three convenience functions. Function `println` is like `print`, but it writes a string followed by a newline. Functions `eprint` and `eprintln` are analogous to `print` and `println`, but they write to standard error. More sophisticated printing functions (Section 1.6.1, page 46) would be lovely, but making such functions type-safe requires code that beginning ML programmers would find baffling.

S215b. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁S215a S215c▷

```
fun println  s = (print s; print "\n")
fun eprint   s = TextIO.output (TextIO.stdErr, s)
fun eprintln s = (eprint s; eprint "\n")
```

To help you diagnose problems that may arise if you decide to implement type checking, type inference, or large integers, I also provide a function for reporting errors that are detected while reading predefined functions.

S215c. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁S215b S215d▷

```
fun predefinedFunctionError s =
  eprintln ("while reading predefined functions, " ^ s)
```

Not all printing is directed to standard output or standard error. To implement the `check-print` unit test (μSmalltalk, Chapter 10), printing must be directed to a buffer. Functions `xprint` and `xprintln` direct output either to standard output or to a buffer, depending on what printing function is currently stored in the mutable reference cell `xprinter`.

S215d. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁S215c S216a▷

```
val xprinter = ref print
fun xprint   s = !xprinter s
fun xprintln s = (xprint s; xprint "\n")
```

The printing function that is stored in `xprinter` can be changed temporarily by calling function `withXprinter`. This function changes `xprinter` just for the duration of a call to `f x`. To restore `xprinter`, function `withXprinter` uses `tryFinally`, which ensures that its `post` handler is always run, even if an exception is raised.

S216a. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁S215d S216b▷

```
withXprinter : (string -> unit) -> ('a -> 'b) -> ('a -> 'b)
tryFinally   : ('a -> 'b) -> 'a -> (unit -> unit) -> 'b
```

```
fun tryFinally f x post =
  (f x handle e => (post (); raise e)) before post ()

fun withXprinter xp f x =
  let val oxp = !xprinter
      val ()  = xprinter := xp
  in  tryFinally f x (fn () => xprinter := oxp)
  end
```

And the function stored in `xprinter` might be `bprint`, which "prints" by appending a string to a buffer. Function `bprinter` returns a pair that contains both `bprint` and a function used to recover the contents of the buffer.

S216b. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁S216a S216c▷

```
fun bprinter () =
  let val buffer = ref []
      fun bprint s = buffer := s :: !buffer
      fun contents () = concat (rev (!buffer))
  in  (bprint, contents)
  end
```

Function `xprint` is used by function `printUTF8`, which prints a Unicode character using the Unicode Transfer Format (UTF-8).

S216c. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁S216b S217a▷

```
fun printUTF8 code =
  let val w = Word.fromInt code
      val (&, >>) = (Word.andb, Word.>>)
      infix 6 & >>
      val _ = if (w & 0wx1fffff) <> w then
                raise RuntimeError (intString code ^
                                    " does not represent a Unicode code point")
              else
                 ()
      val printbyte = xprint o str o chr o Word.toInt
      fun prefix byte byte' = Word.orb (byte, byte')
  in  if w > 0wxffff then
        app printbyte [ prefix 0wxf0  (w >> 0w18)
                      , prefix 0wx80 ((w >> 0w12) & 0wx3f)
                      , prefix 0wx80 ((w >>  0w6) & 0wx3f)
                      , prefix 0wx80 ((w      ) & 0wx3f)
                      ]
      else if w > 0wx7ff then
        app printbyte [ prefix 0wxe0  (w >> 0w12)
                      , prefix 0wx80 ((w >>  0w6) & 0wx3f)
                      , prefix 0wx80 ((w        ) & 0wx3f)
                      ]
      else if w > 0wx7f then
        app printbyte [ prefix 0wxc0  (w >>  0w6)
                      , prefix 0wx80 ((w        ) & 0wx3f)
                      ]
      else
        printbyte w
  end
```

H.2.3 *Utility functions for renaming variables*

In the theory of programming languages, it's common to talk about *fresh names*, where "fresh" means "different from any name in the program or its environment" (Chapters 6 to 8). And if you implement a type checker for a polymorphic language like Typed μScheme, or if you implement type inference, or if you ever implement the lambda calculus, you will need code that generates fresh names. You can always try names like `t1`, `t2`, and so on. But if you want to debug, it's usually helpful to relate the fresh name to a name already in the program. I like to do this by tacking on a numeric suffix; for example, to get a fresh name that's like `x`, I might try `x-1`, `x-2`, and so on. But if the process iterates, I don't want to generate a name like `x-1-1-1`; I'd much rather generate `x-3`. This utility function helps by stripping off any numeric suffix to recover the original `x`.

S217a. ⟨*utility functions for string manipulation and printing* S214c⟩+≡ (S213a) ◁S216c

```
fun stripNumericSuffix s =
      let fun stripPrefix []         = s   (* don't let things get empty *)
            | stripPrefix (#"-"::[]) = s
            | stripPrefix (#"-"::cs) = implode (reverse cs)
            | stripPrefix (c   ::cs) = if Char.isDigit c then stripPrefix cs
                                       else implode (reverse (c::cs))
      in  stripPrefix (reverse (explode s))
      end
```

H.2.4 *Utility functions for sets, collections, and lists*

Sets

Quite a few analyses of programs, including a type checker in Chapter 6 and the type inference in Chapter 7, need to manipulate sets of variables. In small programs, such sets are usually small, so I provide a simple implementation that represents a set using a list with no duplicate elements. It's essentially the same implementation that you see in μScheme in Chapter 2.[1]

S217b. ⟨*simple implementations of set operations* S217b⟩≡ (S213a)

```
type 'a set
emptyset : 'a set
member   : ''a -> ''a set -> bool
insert   : ''a     * ''a set  -> ''a set
union    : ''a set * ''a set  -> ''a set
inter    : ''a set * ''a set  -> ''a set
diff     : ''a set * ''a set  -> ''a set
```

```
type 'a set = 'a list
val emptyset = []
fun member x =
  List.exists (fn y => y = x)
fun insert (x, ys) =
  if member x ys then ys else x::ys
fun union (xs, ys) = foldl insert ys xs
fun inter (xs, ys) =
  List.filter (fn x => member x ys) xs
fun diff  (xs, ys) =
  List.filter (fn x => not (member x ys)) xs
```

[1] The ML types of the set operations include type variables with double primes, like `''a`. The type variable `''a` can be instantiated only with an "equality type." Equality types include base types like strings and integers, as well as user-defined types that do not contain functions. Functions *cannot* be compared for equality.

In the functions above, a set has the same representation as a list, and they can be used interchangeably. Sometimes, however, the thing you're collecting is itself a set, and you want to distinguish the two forms of set (for an example, see Exercise 38 on page 520). For that purpose, I define a type `collection` that is distinct from the set/list type.

S218a. ⟨*collections with mapping and combining functions* S218a⟩≡ (S213a) S218b ▷

```
type 'a collection
elemsC  : 'a collection -> 'a set
singleC : 'a -> 'a collection
emptyC  :       'a collection
```

```
datatype 'a collection = C of 'a set
fun elemsC (C xs) = xs
fun singleC x     = C [x]
val emptyC        = C []
```

The `collection` type is intended to be used some more functions that are defined below. In particular, functions `joinC` and `mapC`, together with `singleC`, form a *monad*. (If you've heard of monads, you may know that they are a useful abstraction for containers and collections of all kinds; they also have more exotic uses, such as expressing input and output as pure functions. The `collection` type is the monad for nondeterminism, which is to say, all possible combinations or outcomes. If you know about monads, you may have picked up some programming tricks you can reuse. But you don't need to know monads to do any of the exercises in this book.)

The key functions on collections are as follows:

- Functions `mapC` and `filterC` do for collections what `map` and `filter` do for lists.
- Function `joinC` takes a collection of collections of τ's and reduces it to a single collection of τ's. When `mapC` is used with a function that itself returns a collection, `joinC` usually follows, as exemplified in the implementation of `mapC2` below.
- Function `mapC2` is the most powerful of all—its type resembles the type of Standard ML's `ListPair.map`, but it works differently: where `ListPair.map` takes elements pairwise, `mapC2` takes all possible combinations. In particular, if you give `ListPair.map` two lists containing N and M elements respectively, the number of elements in the result is $\min(N, M)$. If you give collections of size N and M to `mapC2`, the number of elements in the result is $N \cdot M$.

S218b. ⟨*collections with mapping and combining functions* S218a⟩+≡ (S213a) ◁ S218a

```
joinC   : 'a collection collection -> 'a collection
mapC    : ('a -> 'b)     -> ('a collection -> 'b collection)
filterC : ('a -> bool)   -> ('a collection -> 'a collection)
mapC2   : ('a * 'b -> 'c) -> ('a collection * 'b collection -> 'c collection)
```

```
fun joinC     (C xs) = C (List.concat (map elemsC xs))
fun mapC  f   (C xs) = C (map f xs)
fun filterC p (C xs) = C (List.filter p xs)
fun mapC2 f (xc, yc) = joinC (mapC (fn x => mapC (fn y => f (x, y)) yc) xc)
```

List utilities

Most of the list utilities anyone would need are part of the initial basis of Standard ML. But the type checker for pattern matching in Chapter 8 sometimes needs to unzip a list of triples into a triple of lists. I define unzip3 and also the corresponding zip3.

S219a. ⟨*list functions not provided by Standard ML's initial basis* S219a⟩≡ (S213a) S219b ▷

```
unzip3 : ('a * 'b * 'c) list -> 'a list * 'b list * 'c list
zip3   : 'a list * 'b list * 'c list -> ('a * 'b * 'c) list
```

```
fun unzip3 [] = ([], [], [])
  | unzip3 (trip::trips) =
      let val (x,  y,  z)  = trip
          val (xs, ys, zs) = unzip3 trips
      in  (x::xs, y::ys, z::zs)
      end

fun zip3 ([], [], []) = []
  | zip3 (x::xs, y::ys, z::zs) = (x, y, z) :: zip3 (xs, ys, zs)
  | zip3 _ = raise ListPair.UnequalLengths
```

Standard ML's list-reversal function is called rev, but in this book I use reverse.

S219b. ⟨*list functions not provided by Standard ML's initial basis* S219a⟩+≡ (S213a) ◁ S219a S219c ▷

```
val reverse = rev
```

Function optionList inspects a list of optional values, and if every value is actually present (made with SOME), then it returns the values. Otherwise it returns NONE.

S219c. ⟨*list functions not provided by Standard ML's initial basis* S219a⟩+≡ (S213a) ◁ S219b

```
optionList : 'a option list -> 'a list option
```

```
fun optionList [] = SOME []
  | optionList (NONE :: _) = NONE
  | optionList (SOME x :: rest) =
      (case optionList rest
         of SOME xs => SOME (x :: xs)
          | NONE    => NONE)
```

H.2.5 Duplicate names and internal errors

In bridge-language binding constructs, like lambda, duplicate names are treated as errors. Such names are detected by function duplicatename. If a name x occurs more than twice on a list, duplicatename returns SOME x; otherwise it returns NONE.

S219d. ⟨*support for names and environments* S219d⟩≡ (S213a)

```
duplicatename : name list -> name option
```

```
fun duplicatename [] = NONE
  | duplicatename (x::xs) =
      if List.exists (fn x' => x' = x) xs then
        SOME x
      else
        duplicatename xs
```

type name 303

Some errors might be caused not by a fault in a user's code but in my interpreter code. Such faults are signaled by the InternalError exception.

S219e. ⟨*support for detecting and signaling errors detected at run time* S219e⟩≡ (S213a)

```
exception InternalError of string (* bug in the interpreter *)
```

Raising InternalError is the equivalent of an assertion failure in a language like C.

I must not confuse InternalError with RuntimeError. When the interpreter raises RuntimeError, it means that a user's program got stuck: evaluation led to a state in which the operational semantics couldn't make progress. The fault is the user's. But when the interpreter raises `InternalError`, it means there is a fault in *my* code; the user's program is blameless.

H.2.6 Control by environment variable

Environment variable `BPCOPTIONS`, if set, contains a comma-separated list of flags like `throttle` or `norun`. Function `hasOption` detects if a flag is present.

S220a. ⟨*ability to interrogate environment variable* `BPCOPTIONS` S220a⟩≡ (S213a)

`hasOption : string -> bool`

```
local
  val split = String.tokens (fn c => c = #",")
  val optionTokens =
    getOpt (Option.map split (OS.Process.getEnv "BPCOPTIONS"), [])
in
  fun hasOption s = member s optionTokens
end
```

H.2.7 Utility functions for limiting computation

Each interpreter is supplied with two ways of stopping a runaway computation:

- A *recursion limit* halts the computation if its call stack gets deeper than 6,000 calls.
- A supply of *evaluation fuel* halts the computation after a million calls to `eval`. That's enough to compute the 25th Catalan number in μSmalltalk, for example.

If environment variable `BPCOPTIONS` includes the string `nothrottle`, evaluation fuel is ignored.

S220b. ⟨*function application with overflow checking* S220b⟩≡ (S213a) S221a ▷

```
local
  val defaultRecursionLimit = 6000
  val recursionLimit = ref defaultRecursionLimit
  datatype checkpoint = RECURSION_LIMIT of int

  val evalFuel = ref 1000000
  val throttleCPU = not (hasOption "nothrottle")
in
  (* manipulate recursion limit *)
  fun checkpointLimit () = RECURSION_LIMIT (!recursionLimit)
  fun restoreLimit (RECURSION_LIMIT n) = recursionLimit := n

  (* work with fuel *)
  val defaultEvalFuel = ref (!evalFuel)
  fun fuelRemaining () = !evalFuel
  fun withFuel n f x =
    let val old = !evalFuel
        val _ = evalFuel := n
    in  (f x before evalFuel := old) handle e => (evalFuel := old; raise e)
    end
```

S221a. ⟨*function application with overflow checking* S220b⟩+≡ (S213a) ◁ S220b

```
    (* convert function `f` to respect computation limits *)
    fun applyWithLimits f =
      if !recursionLimit <= 0 then
        ( recursionLimit := defaultRecursionLimit
        ; raise RuntimeError "recursion too deep"
        )
      else if throttleCPU andalso !evalFuel <= 0 then
        ( evalFuel := !defaultEvalFuel
        ; raise RuntimeError "CPU time exhausted"
        )
      else
        let val _ = recursionLimit := !recursionLimit - 1
            val _ = evalFuel       := !evalFuel - 1
        in  fn arg => f arg before (recursionLimit := !recursionLimit + 1)
        end
    fun resetComputationLimits () = ( recursionLimit := defaultRecursionLimit
                                    ; evalFuel := !defaultEvalFuel
                                    )
  end
```

H.3 Representing error outcomes as values

When an error occurs, especially during evaluation, the best and most convenient thing to do is often to raise an ML exception, which can be caught in a handler. But it's not always easy to put a handler exactly where it's needed. To get the code right, it may be better to represent an error outcome as a value. Like any other value, such a value can be passed and returned until it reaches a place where a decision is made.

- When representing the outcome of a unit test, an error means failure for `check-expect` but success for `check-error`. Rather than juggle "exception" versus "non-exception," I treat both outcomes on the same footing, as values. Successful evaluation to produce bridge-language value v is represented as ML value `OK` v. Evaluation that signals an error with message m is represented as ML value `ERROR` m. Constructors `OK` and `ERROR` are the value constructors of the algebraic data type `error`, defined here:

  S221b. ⟨*support for representing errors as ML values* S221b⟩≡ (S213a) S222a ▷

  ```
  datatype 'a error = OK of 'a | ERROR of string
  ```

- My parsers, which use technology described in Appendix I below, are clear and easy to write, but their execution is hopelessly simple-minded. For example, when trying to read an expression, my parser is continually posing very simple questions to its input: Are you an `if`? Are you a `while`? Are you a `set`? And so on. But although the questions are simple, the answers are not. Each question, like the `if` question for example, can be answered three ways:

  - I'm an `if`, and here's my abstract-syntax tree e.
  - I'm not an `if`.
  - I thought I was an `if`, but something went wrong—I must be a syntax error.

`member` S217b
`RuntimeError` S213b

An example of each case is shown here:

```
-> (if (< it 0) 'negative 'nonnegative)     ; I'm an if
nonnegative
-> (+ 2 2)                                  ; I'm not an if
4
-> (if (symbol? it) 99)                     ; I'm a syntax error
syntax error: expected (if e1 e2 e3)
```

If I tried to signal the error case with an exception, I would find it very difficult to build parsers that actually work—and I would continually be worrying about uncaught exceptions. Instead, I represent each form of answer as follows:

- An answer of the form "I'm what you asked for, and here is my abstract-syntax tree e" is represented roughly as `SOME (OK` e`)`.[2]
- An answer of the form "I'm not what you asked for" is represented as `NONE`.
- An answer of the form "I thought I was what you asked for, but something went wrong—I must be a syntax error" is represented roughly as `SOME (ERROR` m`)`, where m is an error message.

Functions that return values like this can be composed using higher-order functions described below.

What if we have a function `f` that could return an `'a` or an error, and another function `g` that expects an `'a`? Because the expression `g (f x)` isn't well typed, standard function composition doesn't exactly make sense, but the *idea* of composition is good. Composition just needs to take a new form, and luckily, there's already a standard. The standard composition relies on a sequencing operator written `>>=`, which uses a special form of continuation-passing style. (The `>>=` operator is traditionally called "bind," but you might wish to pronounce it "and then.") The idea is to apply `f` to `x`, and if the result is `OK y`, to continue by applying `g` to `y`. But if the result of applying `(f x)` is an error, that error is the result of the whole computation. The `>>=` operator sequences the possibly erroneous result `(f x)` with the continuation `g`, so where we might wish to write `g (f x)`, we instead write

```
f x >>= g.
```

In the definition of `>>=`, I write the second function as `k`, not `g`, because `k` is a traditional metavariable for a continuation.

S222a. ⟨*support for representing errors as ML values* S221b⟩+≡ (S213a) ◁S221b S222b▷

```
infix 1 >>=                          >>= : 'a error * ('a -> 'b error) -> 'b error
fun (OK x)      >>= k  =  k x
  | (ERROR msg) >>= k  =  ERROR msg
```

A very common special case occurs when the continuation always succeeds; that is, the continuation `k'` has type `'a -> 'b` instead of `'a -> b error`. In this case, the execution plan is that when `(f x)` succeeds, continue by applying `k'` to the result; otherwise propagate the error. I know of no standard way to write this operator,[3], so I use `>>=+`, which you might also choose to pronounce "and then."

S222b. ⟨*support for representing errors as ML values* S221b⟩+≡ (S213a) ◁S222a S223a▷

```
infix 1 >>=+                         >>=+ : 'a error * ('a -> 'b) -> 'b error
fun e >>=+ k'  =  e >>= (OK o k')
```

[2] "Roughly" because in truth, the answer also includes unread input.

[3] Haskell uses `flip fmap`.

Sometimes I map an error-producing function over a list of values to get a list of `'a error` results. Such a list is hard to work with, and the right thing to do with it is to convert it to a single value that's either an `'a list` or an error. In my code, the conversion operation is called `errorList`.[4] It is implemented by folding over the list of possibly erroneous results, concatenating *all* error messages.

S223a. ⟨*support for representing errors as ML values* S221b⟩+≡ (S213a) ◁S222b S223b▷

```
errorList : 'a error list -> 'a list error
```

```
fun errorList es =
  let fun cons (OK x, OK xs) = OK (x :: xs)
        | cons (ERROR m1, ERROR m2) = ERROR (m1 ^ "; " ^ m2)
        | cons (ERROR m, OK _) = ERROR m
        | cons (OK _, ERROR m) = ERROR m
  in  foldr cons (OK []) es
  end
```

Finally, I sometimes want to label an error message with a string `s`, which can identify where the error originates:

S223b. ⟨*support for representing errors as ML values* S221b⟩+≡ (S213a) ◁S223a

```
fun errorLabel s (OK x) = OK x
  | errorLabel s (ERROR msg) = ERROR (s ^ msg)
```

These functions are used in parsing and elsewhere.

H.4 Unit testing

When running a unit test, each interpreter has to account for the possibility that evaluating an expression causes a run-time error. Just as in Chapters 1 and 2, such an error shouldn't result in an error message; it should just cause the test to fail. (Or if the test expects an error, it should cause the test to succeed.) To manage errors in C, each interpreter had to fool around with `set_error_mode`. In ML, things are simpler: the result of an evaluation is converted either to `OK` v, where v is a value, or to `ERROR` m, where m is an error message, as described above. To use this representation, I define some utility functions.

When a `check-expect` fails, function `whatWasExpected` reports what was expected. If the thing expected was a syntactic value, `whatWasExpected` shows just the value. Otherwise it shows the syntax, plus whatever the syntax evaluated to. The definition of `asSyntacticValue` is language-dependent.

S223c. ⟨*shared* `whatWasExpected` S223c⟩≡ (S224d)

```
whatWasExpected  : exp * value error -> string
asSyntacticValue : exp -> value option
```

```
fun whatWasExpected (e, outcome) =
  case asSyntacticValue e
    of SOME v => valueString v
     | NONE =>
         case outcome
           of OK v => valueString v ^ " (from evaluating " ^ expString e ^ ")"
            | ERROR _ =>  "the result of evaluating " ^ expString e
```

[4] Haskell calls it sequence.

Function `checkExpectPassesWith` runs a `check-expect` test and uses the given `equals` to tell if the test passes. If the test does not pass, `checkExpectPasses` also writes an error message. Error messages are written using `failtest`, which, after writing the error message, indicates failure by returning `false`.

S224a. ⟨*shared* `checkExpectPassesWith`*, which calls* `outcome` S224a⟩≡ (S224d)

```
checkExpectPassesWith : (value * value -> bool) -> exp * exp -> bool
outcome  : exp -> value error
failtest : string list -> bool
```

```
val cxfailed = "check-expect failed: "
fun checkExpectPassesWith equals (checkx, expectx) =
  case (outcome checkx, outcome expectx)
    of (OK check, OK expect) =>
         equals (check, expect) orelse
         failtest [cxfailed, " expected ", expString checkx,
                   " to evaluate to ", whatWasExpected (expectx, OK expect),
                   ", but it's ", valueString check, "."]
     | (ERROR msg, tried) =>
         failtest [cxfailed, " expected ", expString checkx,
                   " to evaluate to ", whatWasExpected (expectx, tried),
                   ", but evaluating ", expString checkx,
                   " caused this error: ", msg]
     | (_, ERROR msg) =>
         failtest [cxfailed, " expected ", expString checkx,
                   " to evaluate to ", whatWasExpected (expectx, ERROR msg),
                   ", but evaluating ", expString expectx,
                   " caused this error: ", msg]
```

Function `checkAssertPasses` does the analogous job for `check-assert`.

S224b. ⟨*shared* `checkAssertPasses` *and* `checkErrorPasses`*, which call* `outcome` S224b⟩≡ (S224d) S224c ▷

```
checkAssertPasses : exp -> bool
```

```
val cafailed = "check-assert failed: "
fun checkAssertPasses checkx =
      case outcome checkx
        of OK check =>
             projectBool check orelse
             failtest [cafailed, " expected assertion ", expString checkx,
                       " to hold, but it doesn't"]
         | ERROR msg =>
             failtest [cafailed, " expected assertion ", expString checkx,
                       " to hold, but evaluating it caused this error: ", msg]
```

Function `checkErrorPasses` does the analogous job for `check-error`.

S224c. ⟨*shared* `checkAssertPasses` *and* `checkErrorPasses`*, which call* `outcome` S224b⟩+≡ (S224d) ◁ S224b

```
checkErrorPasses : exp -> bool
```

```
val cefailed = "check-error failed: "
fun checkErrorPasses checkx =
      case outcome checkx
        of ERROR _ => true
         | OK check =>
             failtest [cefailed, " expected evaluating ", expString checkx,
                       " to cause an error, but evaluation produced ",
                       valueString check]
```

All three functions are stitched together, using a language-dependent `testEquals`.

S224d. ⟨*shared* `check{Expect,Assert,Error{Passes`*, which call* `outcome` S224d⟩≡ (S382c)

⟨*shared* `whatWasExpected` S223c⟩
⟨*shared* `checkExpectPassesWith`*, which calls* `outcome` S224a⟩
⟨*shared* `checkAssertPasses` *and* `checkErrorPasses`*, which call* `outcome` S224b⟩

```
fun checkExpectPasses (cx, ex) = checkExpectPassesWith testEquals (cx, ex)
```

Table H.1: Unit-testing functions

Unit-testing functions provided by each language	
`outcome`	`: exp -> value error`
`ty`	`: exp -> ty error`
`testEquals`	`: value * value -> bool`
`asSyntacticValue`	`: exp -> value option`
`valueString`	`: value -> string`
`expString`	`: exp -> string`
`testIsGood`	`: unit_test list * basis -> bool`

Shared functions for unit testing	
`whatWasExpected`	`: exp * value error -> string`
`checkExpectPasses`	`: exp * exp -> bool`
`checkErrorPasses`	`: exp -> bool`
`numberOfGoodTests`	`: unit_test list * basis -> int`
`processTests`	`: unit_test list * basis -> unit`

The `failtest` used to write the error messages, which always returns `false`, is defined as follows:

S225a. ⟨*shared unit-testing utilities* S225a⟩≡ (S380b) S225b ▷

```
fun failtest strings =
  (app eprint strings; eprint "\n"; false)
```

`failtest : string list -> bool`

In each bridge language, test results are reported the same way. The report's format is stolen from the DrRacket programming environment. If there are no tests, there is no report.

S225b. ⟨*shared unit-testing utilities* S225a⟩+≡ (S380b) ◁ S225a

```
fun reportTestResultsOf what (npassed, nthings) =
  case (npassed, nthings)
    of (_, 0) => ()  (* no report *)
     | (0, 1) => println ("The only " ^ what ^ " failed.")
     | (1, 1) => println ("The only " ^ what ^ " passed.")
     | (0, 2) => println ("Both " ^ what ^ "s failed.")
     | (1, 2) => println ("One of two " ^ what ^ "s passed.")
     | (2, 2) => println ("Both " ^ what ^ "s passed.")
     | _ => if npassed = nthings then
              app print ["All ", intString nthings, " " ^ what ^ "s passed.\n"]
            else if npassed = 0 then
              app print ["All ", intString nthings, " " ^ what ^ "s failed.\n"]
            else
              app print [intString npassed, " of ", intString nthings,
                         " " ^ what ^ "s passed.\n"]
val reportTestResults = reportTestResultsOf "test"
```

Function `processTests` is shared among all bridge languages. For each test, it calls the language-dependent `testIsGood`, adds up the number of good tests, and reports the result.

S226. ⟨*shared definition of* `processTests` S226⟩≡ (S380b)

```
processTests : unit_test list * basis -> unit
```

```
fun processTests (tests, rho) =
      reportTestResults (numberOfGoodTests (tests, rho), length tests)
and numberOfGoodTests (tests, rho) =
  foldr (fn (t, n) => if testIsGood (t, rho) then n + 1 else n) 0 tests
```

H.5 Polymorphic streams, with optional side effects

Every interpreter needs to read input, which is structured as a sequence of lines. And in ML, as in C, a sequence of input lines is available only by executing imperative code. In C, the imperative library function is `fgets`, with which I define `getline_`. In ML, the imperative library function is `TextIO.inputLine`. In both languages, once the line has been gotten, it's gone, and it can't be gotten again. But in a functional language, we might like to treat input lines not as a procession of events in time, but as an actual data structure.

The right data structure isn't a list, because to compute a list would require reading *all* of the input before processing *any* of it. And an interactive interpreter doesn't just read all the input and then convert it all to definitions. Instead, it reads only as much input as is needed to make the first definition, then evaluates the definition and prints the result. The reading and evaluation can be orchestrated using *streams*.

A stream is a data structure that holds the result of a sequence of imperative actions. The results of earlier actions can be inspected before any later actions are performed. By hiding the action of reading behind a stream abstraction, code can treat an input as an immutable sequence of lines... or characters... or extended definitions. In some interpreters, a stream is even used to provide an infinite supply of fresh variables. A stream puts ephemeral results of unrepeatable actions into a data structure that can be kept in memory as long as we like and can be examined as many times as we like.

Streams, like lists, are a powerful abstraction that admits of sophisticated manipulation via higher-order functions, including some of the same functions we use on lists. The stream-related functions defined below are listed in Table H.2 (page S227).

H.5.1 Suspensions: repeatable access to the result of one action

Streams are built around a single abstraction: the *suspension*, which is also called a *thunk*. A suspension of type `'a susp` represents a value of type `'a` that is produced by an action, like reading a line of input. The action is not performed until the suspension's value is *demanded* by function `demand`.[5] The action itself is represented by a function of type `unit -> 'a`. A suspension is created by passing an action to the function `delay`; at that point, the action is "pending." If `demand` is never called, the action is never performed and remains pending. The first time `demand` is called, the

[5] If you're familiar with suspensions or with lazy computation in general, you know that the function `demand` is traditionally called `force`. But I use the name `force` to refer to a similar function in the μHaskell interpreter, which implements a full language around the idea of lazy computation. It is possible to have two functions called `force`—they can coexist peacefully—but I think it's too confusing. So the less important function, which is presented here, is called `demand`. Even though my μHaskell chapter never made it into the book.

Table H.2: Stream-related types and functions

Suspensions	
`type 'a susp`	
`delay`	`: (unit -> 'a) -> 'a susp`
`demand`	`: 'a susp -> 'a`
Polymorphic streams and stream functions	
`type 'a stream`	
`streamGet`	`: 'a stream -> ('a * 'a stream) option`
`streamOfList`	`: 'a list -> 'a stream`
`listOfStream`	`: 'a stream -> 'a list`
`delayedStream`	`: (unit -> 'a stream) -> 'a stream`
`streamOfEffects`	`: (unit -> 'a option) -> 'a stream`
`streamRepeat`	`: 'a -> 'a stream`
`streamOfUnfold`	`: ('b -> ('a * 'b) option) -> 'b -> 'a stream`
`preStream`	`: (unit -> unit) * 'a stream -> 'a stream`
`postStream`	`: 'a stream * ('a -> unit) -> 'a stream`
`streamMap`	`: ('a -> 'b) -> 'a stream -> 'b stream`
`streamFilter`	`: ('a -> bool) -> 'a stream -> 'a stream`
`streamFold`	`: ('a * 'b -> 'b) -> 'b -> 'a stream -> 'b`
`streamZip`	`: 'a stream * 'b stream -> ('a * 'b) stream`
`streamConcat`	`: 'a stream stream -> 'a stream`
`streamConcatMap`	`: ('a -> 'b stream) -> 'a stream -> 'b stream`
`@@@`	`: 'a stream * 'a stream -> 'a stream`
`streamTake`	`: int * 'a stream -> 'a list`
`streamDrop`	`: int * 'a stream -> 'a list`
Streams of numbers, lines, or extended definitions	
`type line = string`	
`type xdef`	
`naturals`	`: int stream`
`filelines`	`: TextIO.instream -> line stream`
`xdefstream`	`: string * line stream * prompts -> xdef stream`
`filexdefs`	`: string * TextIO.instream * prompts -> xdef stream`
`stringsxdefs`	`: string * string list -> xdef stream`

type `basis`,
in μML S439a
in μScheme S238c
in μSmalltalk S548e
in Molecule S481c
in nano-ML S421a
in Typed Impcore S392a
in Typed μScheme 382b
`reportTestResults` S225b
`testIsGood`,
in μHaskell S426a
in μML S447a
in μProlog S595a
in μScheme S382c
in μSmalltalk S571a
in Molecule S542a
in Typed Impcore S395a
in Typed μScheme S409a
type `unit_test`,
in μHaskell S419c
in μML S438c
in μProlog S55b
in μSmalltalk S214a
in Molecule S406a
in Typed Impcore 333b

action is performed, and the suspension saves the result that is produced. If `demand` is called multiple times, the action is still performed just once—later calls to `demand` don't repeat the action; instead they return the value previously produced.

To implement suspensions, I use a standard combination of imperative and functional code. A suspension is a reference to an `action`, which can be pending or can have produced a result.

S228a. ⟨*suspensions* S228a⟩≡ (S213a) S228b ▷

```
type 'a susp
```

```
datatype 'a action
  = PENDING  of unit -> 'a
  | PRODUCED of 'a

type 'a susp = 'a action ref
```

Functions `delay` and `demand` convert to and from suspensions.

S228b. ⟨*suspensions* S228a⟩+≡ (S213a) ◁ S228a

```
delay  : (unit -> 'a) -> 'a susp
demand : 'a susp -> 'a
```

```
fun delay f = ref (PENDING f)
fun demand cell =
  case !cell
    of PENDING f =>  let val result = f ()
                     in  (cell := PRODUCED result; result)
                     end
     | PRODUCED v => v
```

H.5.2 Streams: results of a sequence of actions

A stream behaves much like a list, except that the first time an element is inspected, an action might be taken. And unlike a list, a stream can be infinite. My code uses streams of lines, streams of characters, streams of definitions, and even streams of source-code locations. In this section I define streams and many related utility functions. Most of the utility functions are inspired by list functions like `map`, `filter`, `concat`, `zip`, and `foldl`.

Stream representation and basic functions

The representation of a stream takes one of three forms:[6]

- The `EOS` constructor represents an empty stream.
- The `:::` constructor (pronounced "cons"), which should remind you of ML's `::` constructor for lists, represents a stream in which an action has already been taken, and the first element of the stream is available (as are the remaining elements). Like the `::` constructor for lists, the `:::` constructor is written as an infix operator.
- The `SUSPENDED` constructor represents a stream in which the action needed to produce the next element may not yet have been taken. Getting the element requires demanding a value from a suspension, and if the action in the suspension is pending, it is performed at that time.

S228c. ⟨*streams* S228c⟩≡ (S213a) S229a ▷

```
datatype 'a stream
  = EOS
  | :::       of 'a * 'a stream
  | SUSPENDED of 'a stream susp
infixr 3 :::
```

[6] There are representations that use fewer forms, but this one has the merit that I can define a polymorphic empty stream without running afoul of ML's "value restriction."

Even though its representation uses mutable state (the suspension), the stream is an immutable abstraction.[7] To observe that abstraction, call `streamGet`. This function performs whatever actions are needed either to produce a pair holding an element an a stream (represented as `SOME (`x`, `xs`)`) or to decide that the stream is empty and no more elements can be produced (represented as `NONE`).

S229a. ⟨*streams* S228c⟩+≡ (S213a) ◁S228c S229b▷

```
streamGet : 'a stream -> ('a * 'a stream) option
```

```
fun streamGet EOS = NONE
  | streamGet (x ::: xs)    = SOME (x, xs)
  | streamGet (SUSPENDED s) = streamGet (demand s)
```

The simplest way to create a stream is by using the `:::` or `EOS` constructor. A stream can also be created from a list. When such a stream is read, no new actions are performed.

S229b. ⟨*streams* S228c⟩+≡ (S213a) ◁S229a S229c▷

```
streamOfList : 'a list -> 'a stream
```

```
fun streamOfList xs =
  foldr (op :::) EOS xs
```

Function `listOfStream` creates a list from a stream. It is useful for debugging.

S229c. ⟨*streams* S228c⟩+≡ (S213a) ◁S229b S229d▷

```
listOfStream : 'a stream -> 'a list
```

```
fun listOfStream xs =
  case streamGet xs
    of NONE => []
     | SOME (x, xs) => x :: listOfStream xs
```

The more interesting streams are those that result from actions. To help create such streams, I define `delayedStream` as a convenience abbreviation for creating a stream from one action.

S229d. ⟨*streams* S228c⟩+≡ (S213a) ◁S229c S229e▷

```
delayedStream : (unit -> 'a stream) -> 'a stream
```

```
fun delayedStream action =
  SUSPENDED (delay action)
```

Creating streams using actions and functions

Function `streamOfEffects` produces the stream of results obtained by repeatedly performing a single action (like reading a line of input). The action must have type `unit -> 'a option`; the stream performs the action repeatedly, producing a stream of `'a` values until performing the action returns `NONE`.

S229e. ⟨*streams* S228c⟩+≡ (S213a) ◁S229d S229f▷

```
streamOfEffects : (unit -> 'a option) -> 'a stream
```

```
fun streamOfEffects action =
  delayedStream (fn () => case action ()
                            of NONE   => EOS
                             | SOME a => a ::: streamOfEffects action)
```

Function `streamOfEffects` can be used to produce a stream of lines from an input file:

S229f. ⟨*streams* S228c⟩+≡ (S213a) ◁S229e S230a▷

```
type line
filelines : TextIO.instream -> line stream
```

```
type line = string
fun filelines infile =
  streamOfEffects (fn () => TextIO.inputLine infile)
```

[7] When debugging, I sometimes violate the abstraction and look at the state of a `SUSPENDED` stream.

Where `streamOfEffects` produces the results of repeating a single *action* again and again, `streamRepeat` repeats a single *value* again and again. This operation might sound useless, but here's an example: suppose we read a sequence of lines from a file, and for error reporting, we want to tag each line with its source location, i.e., file name and line number. Well, the file names are all the same, and one easy way to associate the same file name with every line is to repeat the file name indefinitely, then join the two streams using `streamZip`. Function `streamRepeat` creates an infinite stream that repeats a value. It works on values of any type.

S230a. ⟨*streams* S228c⟩+≡ (S213a) ◁ S229f S230b ▷

```
streamRepeat : 'a -> 'a stream
```

```
fun streamRepeat x =
  delayedStream (fn () => x ::: streamRepeat x)
```

A more sophisticated way to produce a stream is to use a function that depends on an evolving *state* of some unknown type `'b`. The function is applied to a state (of type `'b`) and may produce a pair containing a value of type `'a` and a new state. Repeatedly applying the function can produce a sequence of results of type `'a`. This operation, in which a function is used to expand a value into a sequence, is the dual of the *fold* operation, which is used to collapse a sequence into a value. The new operation is therefore called *unfold*.

S230b. ⟨*streams* S228c⟩+≡ (S213a) ◁ S230a S230c ▷

```
streamOfUnfold : ('b -> ('a * 'b) option) -> 'b -> 'a stream
```

```
fun streamOfUnfold next state =
  delayedStream
    (fn () => case next state
                of NONE => EOS
                 | SOME (a, state') => a ::: streamOfUnfold next state')
```

Function `streamOfUnfold` can turn any "get" function into a stream. In fact, the unfold and get operations should obey the following algebraic law:

$$\texttt{streamOfUnfold streamGet}\ xs \equiv xs.$$

Another useful "get" function is `(fn n => SOME (n, n+1))`; passing this function to `streamOfUnfold` results in an infinite stream of increasing integers.

S230c. ⟨*streams* S228c⟩+≡ (S213a) ◁ S230b S231a ▷

```
naturals : int stream
```

```
val naturals =
  streamOfUnfold (fn n => SOME (n, n+1)) 0   (* 0 to infinity *)
```

(Streams, like lists, support not only unfolding but also folding. The fold function `streamFold` is defined below in chunk S231e.)

Attaching extra actions to streams

A stream built with `streamOfEffects` or `filelines` has an imperative action built in. But in an interactive interpreter, the action of reading a line should be preceded by another action: printing the prompt. And deciding just what prompt to print requires orchestrating other actions. One option, which I use below, is to attach an imperative action to a "get" function used with `streamOfUnfold`. Another option, which is sometimes easier to understand, is to attach an action to the stream itself. Such an action could reasonably be performed either before or after the action of getting an element from the stream.

Given an action `pre` and a stream xs, I define a stream `preStream (pre,` xs`)` that adds `pre ()` to the action performed by the stream. Roughly speaking,

$$\texttt{streamGet (preStream (pre, }xs\texttt{))} = \texttt{(pre (); streamGet }xs\texttt{)}.$$

(The equivalence is only rough because the `pre` action is performed lazily, only when an action is needed to get a value from xs.)

S231a. ⟨*streams* S228c⟩+≡ (S213a) ◁S230c S231b▷

```
preStream : (unit -> unit) * 'a stream -> 'a stream
```

```
fun preStream (pre, xs) =
  streamOfUnfold (fn xs => (pre (); streamGet xs)) xs
```

It's also useful to be able to perform an action immediately *after* getting an element from a stream. In `postStream`, I perform the action only if `streamGet` succeeds. By performing the `post` action only when `streamGet` succeeds, I make it possible to write a `post` action that has access to the element just gotten. Post-get actions are especially useful for debugging.

S231b. ⟨*streams* S228c⟩+≡ (S213a) ◁S231a S231c▷

```
postStream : 'a stream * ('a -> unit) -> 'a stream
```

```
fun postStream (xs, post) =
  streamOfUnfold (fn xs => case streamGet xs
                             of NONE => NONE
                              | head as SOME (x, _) => (post x; head)) xs
```

Standard list functions ported to streams

Functions like `map`, `filter`, `fold`, `zip`, and `concat` are every bit as useful on streams as they are on lists.

S231c. ⟨*streams* S228c⟩+≡ (S213a) ◁S231b S231d▷

```
streamMap : ('a -> 'b) -> 'a stream -> 'b stream
```

```
fun streamMap f xs =
  delayedStream (fn () => case streamGet xs
                            of NONE => EOS
                             | SOME (x, xs) => f x ::: streamMap f xs)
```

S231d. ⟨*streams* S228c⟩+≡ (S213a) ◁S231c S231e▷

```
streamFilter : ('a -> bool) -> 'a stream -> 'a stream
```

```
fun streamFilter p xs =
  delayedStream (fn () => case streamGet xs
                            of NONE => EOS
                             | SOME (x, xs) => if p x then x ::: streamFilter p xs
                                               else streamFilter p xs)
```

The only sensible order in which to fold the elements of a stream is the order in which the actions are taken and the results are produced: from left to right.

S231e. ⟨*streams* S228c⟩+≡ (S213a) ◁S231d S231f▷

```
streamFold : ('a * 'b -> 'b) -> 'b -> 'a stream -> 'b
```

```
fun streamFold f z xs =
  case streamGet xs of NONE => z
                     | SOME (x, xs) => streamFold f (f (x, z)) xs
```

`:::`	S228c
`delayedStream`	S229d
`EOS`	S228c
`type stream`	S228c
`streamGet`	S229a

Function `streamZip` returns a stream that is as long as the shorter of the two argument streams. In particular, if `streamZip` is applied to a finite stream and an infinite stream, the result is a finite stream.

S231f. ⟨*streams* S228c⟩+≡ (S213a) ◁S231e S232a▷

```
streamZip : 'a stream * 'b stream -> ('a * 'b) stream
```

```
fun streamZip (xs, ys) =
  delayedStream
  (fn () => case (streamGet xs, streamGet ys)
              of (SOME (x, xs), SOME (y, ys)) => (x, y) ::: streamZip (xs, ys)
               | _ => EOS)
```

Concatenation turns a stream of streams of τ's into a single stream of τ's. I define it using a `streamOfUnfold` with a two-part state: the first element of the state holds an initial `xs`, and the second part holds the stream of all remaining streams, `xss`. To concatenate the stream of streams `xss`, I use an initial state of `(EOS, xss)`.

S232a. ⟨*streams* S228c⟩+≡ (S213a) ◁S231f S232b▷

```
streamConcat : 'a stream stream -> 'a stream
```

```
fun streamConcat xss =
  let fun get (xs, xss) =
        case streamGet xs
          of SOME (x, xs) => SOME (x, (xs, xss))
           | NONE => case streamGet xss
                       of SOME (xs, xss) => get (xs, xss)
                        | NONE => NONE
  in  streamOfUnfold get (EOS, xss)
  end
```

In list and stream processing, `concat` is very often composed with `map f`. The composition is usually called `concatMap`.

S232b. ⟨*streams* S228c⟩+≡ (S213a) ◁S232a S232c▷

```
streamConcatMap : ('a -> 'b stream) -> 'a stream -> 'b stream
```

```
fun streamConcatMap f xs = streamConcat (streamMap f xs)
```

The code used to append two streams is much like the code used to concatenate arbitrarily many streams. To avoid duplicating the tricky manipulation of states, I implement append using concatenation.

S232c. ⟨*streams* S228c⟩+≡ (S213a) ◁S232b S232d▷

```
@@@ : 'a stream * 'a stream -> 'a stream
```

```
infix 5 @@@
fun xs @@@ xs' = streamConcat (streamOfList [xs, xs'])
```

Whenever I rename bound variables, for example in a type $\forall \alpha_1, \ldots, \alpha_n \,.\, \tau$, I have to choose new names that don't conflict with existing names in τ or in the environment. The easiest way to get good names to build an infinite stream of names by using `streamMap` on `naturals`, then use `streamFilter` to choose only the good ones, and finally to take exactly as many good names as I need by calling `streamTake`, which is defined here.

S232d. ⟨*streams* S228c⟩+≡ (S213a) ◁S232c S232e▷

```
streamTake : int * 'a stream -> 'a list
```

```
fun streamTake (0, xs) = []
  | streamTake (n, xs) =
      case streamGet xs
        of SOME (x, xs) => x :: streamTake (n-1, xs)
         | NONE => []
```

Once I've used `streamTake`, I get the rest of the stream with `streamDrop` (chunk S235c).

S232e. ⟨*streams* S228c⟩+≡ (S213a) ◁S232d

```
streamDrop : int * 'a stream -> 'a stream
```

```
fun streamDrop (0, xs) = xs
  | streamDrop (n, xs) =
      case streamGet xs
        of SOME (_, xs) => streamDrop (n-1, xs)
         | NONE => EOS
```

H.5.3 *Streams of extended definitions*

Every bridge language has its own parser, called `xdefstream`, which converts a stream of lines to a stream of `xdefs`. But as in Section F.2.3, the convenience functions `filexdefs` and `stringsxdefs` are shared.

S233a. ⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S233a⟩≡ (S383c)

```
xdefstream   : string * line stream     * prompts -> xdef stream
filexdefs    : string * TextIO.instream * prompts -> xdef stream
stringsxdefs : string * string list                -> xdef stream
```

```
fun filexdefs (filename, fd, prompts) =
      xdefstream (filename, filelines fd, prompts)
fun stringsxdefs (name, strings) =
      xdefstream (name, streamOfList strings, noPrompts)
```

H.6 Tracking and reporting source-code locations

An error message is more informative if it says where the error occurred. "Where" means a *source-code location*. Compilers that take themselves seriously report source-code locations right down to the individual character: file `broken.c`, line 12, column 17. In production compilers, such precision is admirable. But in a pedagogical interpreter, precision sometimes gives way to simplicity. A good compromise is to track only source file and line number. That's precise enough to help programmers find errors, and it simplifies the implementation by eliminating the bookkeeping that would otherwise be needed to track column numbers.

S233b. ⟨*support for source-code locations and located streams* S233b⟩≡ (S213a) S233c ▷

```
type srcloc = string * int
fun srclocString (source, line) =
  source ^ ", line " ^ intString line
```

```
type srcloc
srclocString : srcloc -> string
```

Source-code locations are useful when reading code from a file. When reading code interactively, however, a message that says the error occurred "in standard input, line 12," is more annoying than helpful. As in the C code in Section F.5.1 (page S181), I use an *error format* to control when error messages include source-code locations. The format is initially set to include them.

S233c. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁ S233b S233d ▷

```
datatype error_format = WITH_LOCATIONS | WITHOUT_LOCATIONS
val toplevel_error_format = ref WITH_LOCATIONS
```

The format is consulted by function `synerrormsg`, which produces the message that accompanies a syntax error. The source location may be omitted only for standard input; error messages about files loaded with `use` are always accompanied by source-code locations.

S233d. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁ S233c S234a ▷

```
fun synerrormsg (source, line) strings =
  if !toplevel_error_format = WITHOUT_LOCATIONS
  andalso source = "standard input"
  then
    concat ("syntax error: " :: strings)
  else
    concat ("syntax error in " :: srclocString (source, line) :: ": " :: strings)
```

The same format determines how warnings are delivered.

S234a. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁S233d S234b▷

```
fun warnAt (source, line) strings =
  ( app eprint
      (if !toplevel_error_format = WITHOUT_LOCATIONS
       andalso source = "standard input"
       then
         "warning: " :: strings
       else
         "warning in " :: srclocString (source, line) :: ": " :: strings)
  ; eprint "\n"
  )
```

Source locations are also used at run time. Any exception can be marked with a location by converting it to the `Located` exception:

S234b. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁S234a S234c▷

```
exception Located of srcloc * exn
```

To keep track of the source location of a line, token, expression, or other datum, I put the location and the datum together in a pair. To make it easier to read the types, I define a type abbreviation which says that a value paired with a location is "located."

S234c. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁S234b S234d▷

```
type 'a located = srcloc * 'a
```

```
type 'a located
```

The `Located` exception is raised by function `atLoc`. Calling `atLoc f x` applies `f` to `x` within the scope of handlers that convert recognized exceptions to the `Located` exception:

S234d. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁S234c S234f▷

```
atLoc : srcloc -> ('a -> 'b) -> ('a -> 'b)
```

```
fun atLoc loc f a =
  f a handle e as RuntimeError _ => raise Located (loc, e)
           | e as NotFound _     => raise Located (loc, e)
           ⟨more handlers for atLoc S234e⟩
```

In addition to exceptions that I have defined, `atLoc` also recognizes and wraps some of Standard ML's predefined exceptions. Handlers for even more exceptions, like `TypeError`, can be added using Noweb.

S234e. ⟨*more handlers for* `atLoc` S234e⟩≡ (S234d)

```
| e as IO.Io _   => raise Located (loc, e)
| e as Div       => raise Located (loc, e)
| e as Overflow  => raise Located (loc, e)
| e as Subscript => raise Located (loc, e)
| e as Size      => raise Located (loc, e)
```

Function `atLoc` is often called by the higher-order function `located`, which converts a function that expects `'a` into a function that expects `'a located`. Function `leftLocated` does something similar for a pair in which only the left half must include a source-code location.

S234f. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁S234d S235a▷

```
located : ('a -> 'b) -> ('a located -> 'b)
leftLocated : ('a * 'b -> 'c) -> ('a located * 'b -> 'c)
```

```
fun located f (loc, a) = atLoc loc f a
fun leftLocated f ((loc, a), b) = atLoc loc f (a, b)
```

A source-code location can appear anywhere in an error message. To make it easy to write error messages that include source-code locations, I define function `fillComplaintTemplate`. This function replaces the string `"<at loc>"` with a reference to a source-code location—or if there is no source-code location, it strips `"<at loc>"` entirely. The implementation uses Standard ML's `Substring` module.

S235a. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁S234f S235b▷

`fillComplaintTemplate : string * srcloc option -> string`

```
fun fillComplaintTemplate (s, maybeLoc) =
  let val string_to_fill = " <at loc>"
      val (prefix, atloc) =
        Substring.position string_to_fill (Substring.full s)
      val suffix = Substring.triml (size string_to_fill) atloc
      val splice_in =
        Substring.full
          (case maybeLoc
             of NONE => ""
              | SOME (loc as (file, line)) =>
                  if !toplevel_error_format = WITHOUT_LOCATIONS
                  andalso file = "standard input"
                  then
                    ""
                  else
                    " in " ^ srclocString loc)
  in  if Substring.size atloc = 0 then (* <at loc> is not present *)
        s
      else
        Substring.concat [prefix, splice_in, suffix]
  end
fun fillAtLoc (s, loc) = fillComplaintTemplate (s, SOME loc)
fun stripAtLoc s = fillComplaintTemplate (s, NONE)
```

To signal a syntax error at a given location, code calls `synerrorAt`.

S235b. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁S235a S235c▷

`synerrorAt : string -> srcloc -> 'a error`

```
fun synerrorAt msg loc =
  ERROR (synerrormsg loc [msg])
```

All locations originate in a located stream of lines. The locations share a filename, and the line numbers are $1, 2, 3, \ldots$ and so on.

S235c. ⟨*support for source-code locations and located streams* S233b⟩+≡ (S213a) ◁S235b

`locatedStream : string * line stream -> line located stream`

```
fun locatedStream (streamname, inputs) =
  let val locations =
        streamZip (streamRepeat streamname, streamDrop (1, naturals))
  in  streamZip (locations, inputs)
  end
```

`eprint`	S215b
`ERROR`	S221b
type `error`	S221b
`naturals`	S230c
`NotFound`	305b
`RuntimeError`	S213b
type `srcloc`	S233b
`srclocString`	S233b
type `stream`	S228c
`streamDrop`	S232e
`streamRepeat`	S230a
`streamZip`	S231f
`synerrormsg`	S233d
`toplevel_error_format`	S233c
`WITHOUT_LOCATIONS`	S233c

H.7 A reusable read-eval-print loop

In each bridge-language interpreter, functions `eval` and `evaldef` process expressions and true definitions. But each interpreter also has to process the extended definitions `USE` and `TEST`, which need more tooling:

- To process a `USE`, the interpreter must be able to parse definitions from a file and enter a read-eval-print loop recursively.
- To process a `TEST` (like `check_expect` or `check_error`), the interpreter must be able to run tests, and to run a test, it must call `eval`.

Much the tooling can be shared among more than one bridge language. To make sharing easy, I introduce some abstraction.

- Type `basis`, which is different for each bridge language, stands for the collection of environment or environments that are used at top level to evaluate a definition. The name *basis* comes from *The Definition of Standard ML* (Milner et al. 1997).

  For μScheme, a `basis` is a single environment that maps each name to a mutable location holding a value. For Impcore, a `basis` would include both global-variable and function environments. And for later languages that have static types, a `basis` includes environments that store information about types.

- Function `processDef`, which is different for each bridge language, takes a `def` and a `basis` and returns an updated `basis`. For μScheme, `processDef` just evaluates the definition, using `evaldef`. For languages that have static types (Typed Impcore, Typed μScheme, and nano-ML in Chapters 6 and 7, among others), `processDef` includes two phases: type checking followed by evaluation.

  Function `processDef` also needs to be told about interaction, which has two dimensions: input and output. On input, an interpreter may or may not prompt:

  S236a. ⟨*type* `interactivity` *plus related functions and value* S236a⟩≡ (S213a) S236b ▷

  ```
  datatype input_interactivity = PROMPTING | NOT_PROMPTING
  ```

  On output, an interpreter may or may not show a response to each definition.

  S236b. ⟨*type* `interactivity` *plus related functions and value* S236a⟩+≡ (S213a) ◁ S236a S236c ▷

  ```
  datatype output_interactivity = ECHOING | NOT_ECHOING
  ```

  The two of information together form a value of type `interactivity`. Such a value can be queried by predicates `prompts` and `print`.

  S236c. ⟨*type* `interactivity` *plus related functions and value* S236a⟩+≡ (S213a) ◁ S236b

  ```
  type interactivity
  noninteractive : interactivity
  prompts : interactivity -> bool
  echoes  : interactivity -> bool
  ```

  ```
  type interactivity =
    input_interactivity * output_interactivity
  val noninteractive =
    (NOT_PROMPTING, NOT_ECHOING)
  fun prompts (PROMPTING,     _) = true
    | prompts (NOT_PROMPTING, _) = false
  fun echoes (_, ECHOING)     = true
    | echoes (_, NOT_ECHOING) = false
  ```

- Function `testIsGood`, which can be shared among languages that share the same definition of `unit_test`, says whether a test passes (or in a typed language, whether the test is well-typed and passes). Function `testIsGood` has a slightly different interface from the corresponding C function `test_result`. The reasons are discussed in Appendix O on page S382.

These pieces can be used to define a single version of `processTests` (Section H.4, page S226) and a single read-eval-print loop, each of which is shared among many bridge languages. The pieces are organized as follows:

Given processDef and testIsGood, function readEvalPrintWith processes a stream of extended definitions. As in the C version, a stream is created using filexdefs or stringsxdefs.

Function readEvalPrintWith has a type that resembles the type of the C function readevalprint, but the ML version takes an extra parameter errmsg. Using this parameter, I issue a special error message when there's a problem in the initial basis (see function predefinedError on page S215). The special error message helps with some of the exercises in Chapters 6 and 7, where if something goes wrong with the implementation of types, an interpreter could fail while trying to read its initial basis. (Failure while reading the basis can manifest in mystifying ways; the special message demystifies the failure.)

S237. ⟨*shared read-eval-print loop* S237⟩≡ (S380b)

```
type basis
processDef   : def * basis * interactivity -> basis
testIsGood   : unit_test      * basis -> bool
processTests : unit_test list * basis -> unit
readEvalPrintWith : (string -> unit) ->
                    xdef stream * basis * interactivity -> basis
processXDef       : xdef * basis -> basis
```

```
fun readEvalPrintWith errmsg (xdefs, basis, interactivity) =
  let val unitTests = ref []
      ⟨definition of processXDef, which can modify unitTests and call errmsg S238b⟩
      val basis = streamFold processXDef basis xdefs
      val _     = processTests (!unitTests, basis)
  in  basis
  end
```

Function readEvalPrintWith executes essentially the same imperative actions as the C function readevalprint (chunk S310e): allocate space for a list of pending unit tests; loop through a stream of extended definitions, using each one to update the environment(s); and process the pending unit tests. (The looping action in the ML code is implemented by function streamFold, which applies processXDef to every element of xdefs. Function streamFold is the stream analog of the list function foldl.) Unlike the C readevalprint, which updates the environment in place by writing through a pointer, the ML function ends by returning a new basis, which contains the updated environment(s).

Please pause and look at the names of the functions. Functions eval and evaldef are named after a specific, technical action: they *evaluate*. But functions processDef, processXDef, and processTests are named after a vague action: they *process*. I've chosen this vague word deliberately, because the "processing" is different in different languages:

processTests	S226
processXDef	S238b
type stream	S228c
streamFold	S231e
type xdef	S214b

- In an untyped language like μScheme or μSmalltalk, "process" means "evaluate."

- In a typed language like Typed Impcore, Typed μScheme, nano-ML, or μML, "process" means "first typecheck, then evaluate."

Using the vague word "process" to cover both language families helps me write generic code that works with both language families.

Let's see the generic code that "processes" an extended definition. To process a `USE` form, `processXDef` calls function `useFile`, which reads definitions from a file and recursively passes them to `readEvalPrintWith`.

S238a. ⟨*definition of* `useFile`*, to read from a file* S238a⟩≡ (S238b)

```
fun useFile filename =
  let val fd = TextIO.openIn filename
      val (_, printing) = interactivity
      val inter' = (NOT_PROMPTING, printing)
  in  readEvalPrintWith errmsg (filexdefs (filename, fd, noPrompts), basis, inter')
      before TextIO.closeIn fd
  end
```

The extended-definition forms `USE` and `TEST` are implemented in exactly the same way for every language: internal function `try` passes each `USE` to `useFile`, and it adds each `TEST` to the mutable list `unitTests`—just as in the C code in Section 1.6.2 (page 53). Function `try` passes each true definition `DEF` to function `processDef`, which does the language-dependent work.

S238b. ⟨*definition of* `processXDef`*, which can modify* `unitTests` *and call* `errmsg` S238b⟩≡ (S237)

```
errmsg     : string -> unit
processDef : def * basis * interactivity -> basis
```

```
fun processXDef (xd, basis) =
  let ⟨definition of useFile, to read from a file S238a⟩
      fun try (USE filename) = useFile filename
        | try (TEST t)       = (unitTests := t :: !unitTests; basis)
        | try (DEF def)      = processDef (def, basis, interactivity)
      fun caught msg = (errmsg (stripAtLoc msg); basis)
  in  withHandlers try xd caught
  end
```

When processing a definition, `processXDef` must recover from any errors that occur. It uses functions `withHandlers` and `caught`. Calling `withHandlers f a caught` normally applies function `f` to argument `a` and returns the result. But when the application of `f` raises an exception that the interpreter should recover from, `withHandlers` calls `caught` with an appropriate error message. Here, `caught` passes the message to `errmsg`, then returns the original `basis` unchanged.

The language-dependent `basis` is, for μScheme, the single environment ρ, which maps each name to a mutable location that holds a value. The basis is the second parameter to `processDef`, which calls `evaldef`, prints its response, and returns a new basis.

S238c. ⟨*definitions of* `eval`*,* `evaldef`*,* `basis`*, and* `processDef` *for μScheme* S238c⟩≡ (S380b)

```
type basis = value ref env
fun processDef (d, rho, interactivity) =
  let val (rho', response) = evaldef (d, rho)
      val _ = if echoes interactivity then println response else ()
  in  rho'
  end
```

A last word about function `readEvalPrintWith`: you might be wondering, "where does it read, evaluate, and print?" It has helpers for that: reading is a side effect of `streamGet`, which is called by `streamFold`, and evaluating and printing are done by `processDef`. But the function is called `readEvalPrintWith` because when you want reading, evaluating, and printing to happen, you call `readEvalPrintWith eprintln`, passing your extended definitions and your environments.

H.8 Handling exceptions

When an exception is raised, a bridge-language interpreter must "catch" or "handle" it. An exception is caught using a syntactic form written with the keyword `handle`. (This form resembles a combination of a `case` expression with the `try-catch` form from Chapter 3.) Within the `handle`, every exception that the interpreter recognizes is mapped to an error message tailored for that exception. To be sure that every exception is responded to in the same way, no matter where it is handled, I write just a single `handle` form, and I deploy it in a higher-order, continuation-passing function: `withHandlers`.

In normal execution, calling `withHandlers f a caught` applies function `f` to argument `a` and returns the result. But when the application `f a` raises an exception, `withHandlers` uses `handle` to recover from the exception and to pass an error message to `caught`, which acts as a failure continuation (Section 2.10, page 136). Each error message contains the string `"<at loc>"`, which can be removed (by `stripAtLoc`) or can be filled in with an appropriate source-code location (by `fillAtLoc`).

The most important exceptions are `NotFound`, `RuntimeError`, and `Located`. Exception `NotFound` is defined in Chapter 5; the others are defined in this appendix. Exceptions `NotFound` and `RuntimeError` signal problems with an environment or with evaluation, respectively. Exception `Located` wraps *another* exception `exn` in a source-code location. When `Located` is caught, `withHandlers` calls itself recursively with a function that "re-raises" exception `exn` and with a failure continuation that fills in the source location in `exn`'s error message.

S239a. ⟨*shared definition of* `withHandlers` S239a⟩≡ (S380b)

```
withHandlers : ('a -> 'b) -> 'a -> (string -> 'b) -> 'b
```

```
fun withHandlers f a caught =
  f a
  handle RuntimeError msg   => caught ("Run-time error <at loc>: " ^ msg)
       | NotFound x         => caught ("Name " ^ x ^ " not found <at loc>")
       | Located (loc, exn) =>
           withHandlers (fn _ => raise exn)
                        a
                        (fn s => caught (fillAtLoc (s, loc)))
       ⟨other handlers that catch non-fatal exceptions and pass messages to caught S239b⟩
```

In addition to `RuntimeError`, `NotFound`, and `Located`, `withHandlers` catches many exceptions that are predefined ML's Standard Basis Library. These exceptions signal things that can go wrong when evaluating an expression or reading a file.

S239b. ⟨*other handlers that catch non-fatal exceptions and pass messages to* `caught` S239b⟩≡ (S239a)

```
| Div                  => caught ("Division by zero <at loc>")
| Overflow             => caught ("Arithmetic overflow <at loc>")
| Subscript            => caught ("Array index out of bounds <at loc>")
| Size                 => caught ("Array length too large (or negative) <at loc>")
| IO.Io { name, ...}   => caught ("I/O error <at loc>: " ^ name)
```

These exception handlers are used in all the bridge-language interpreters.

H.9 Interpreter setup and command-line processing

In each interpreter, something has to act like the C function `main`. This code has to initialize the interpreter and start evaluating extended definitions.

Part of initialization is setting the global error format. The reusable function `setup_error_format` uses interactivity to set the error format, which, as in the C versions, determines whether syntax-error messages include source-code locations (see functions `synerrorAt` and `synerrormsg` on pages S233 and S235).

S240a. ⟨*shared utility functions for initializing interpreters* S240a⟩≡ (S213a)

```
fun setup_error_format interactivity =
  if prompts interactivity then
    toplevel_error_format := WITHOUT_LOCATIONS
  else
    toplevel_error_format := WITH_LOCATIONS
```

A bridge-language interpreter can be run on standard input or on a named file. Either one can be converted to a stream, so the code that runs an interpreter is defined on a stream, by function `runStream`. This runs the code found in a given, named input, using a given interactivity mode. The interactivity mode determines both the error format and the prompts. Function `runStream` then starts the read-eval-print loop, using the initial basis.

S240b. ⟨*function* `runStream`*, which evaluates input given* `initialBasis` S240b⟩≡ (S379)

```
runStream : string -> TextIO.instream -> interactivity -> basis -> basis
```

```
fun runStream inputName input interactivity basis =
  let val _ = setup_error_format interactivity
      val prompts = if prompts interactivity then stdPrompts else noPrompts
      val xdefs = filexdefs (inputName, input, prompts)
  in  readEvalPrintWith eprintln (xdefs, basis, interactivity)
  end
```

(Function `runStream` gets its own code chunk because the μSmalltalk interpreter needs a slightly different version.)

If files are named on a command line, each file is passed to function `runPathWith`. This function opens the named file and calls `runStream`. And in a special hack, relatively common on Unix systems, the name `-` stands for standard input.

S240c. ⟨*look at command-line arguments, then run* S240c⟩≡ (S379) S240d ▷

```
runPathWith : interactivity -> (string * basis -> basis)
```

```
fun runPathWith interactivity ("-", basis) =
      runStream "standard input" TextIO.stdIn interactivity basis
  | runPathWith interactivity (path, basis) =
      let val fd = TextIO.openIn path
      in  runStream path fd interactivity basis
          before TextIO.closeIn fd
      end
```

If an interpreter doesn't recognize a command-line option, it can print a usage message. A usage-message function needs to know the available options, but each available option is associated with a function that performs an action, and if something goes wrong, the action function might need to call the usage function. I resolve this mutual recursion by first allocating a mutual cell to hold the usage function, then updating it later. This is also how `letrec` is implemented in μScheme.

S240d. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁ S240c S241a ▷

```
val usage = ref (fn () => ())
```

```
usage : (unit -> unit) ref
```

To represent actions that might be called for by command-line options, I define type `action`.

S241a. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁S240d S241b▷

```
datatype action
  = RUN_WITH of interactivity  (* call runPathWith on remaining arguments *)
  | DUMP     of unit -> unit   (* dump information *)
  | FAIL     of string         (* signal a bad command line *)
  | DEFAULT                    (* no command-line options were given *)
```

The default action is to run the interpreter in its most interactive mode.

S241b. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁S241a S241c▷

```
val default_action = RUN_WITH (PROMPTING, ECHOING)
```

An action is performed by function `perform`. Not every action makes sense with arguments.

S241c. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁S241b S241d▷

`perform: action * string list -> unit`

```
fun perform (RUN_WITH interactivity, []) =
      perform (RUN_WITH interactivity, ["-"])
  | perform (RUN_WITH interactivity, args) =
      ignore (foldl (runPathWith interactivity) initialBasis args)
  | perform (DUMP go, [])     = go ()
  | perform (DUMP go, _ :: _) = perform (FAIL "Dump options take no files", [])
  | perform (FAIL msg, _)     = (eprintln msg; !usage())
  | perform (DEFAULT, args)   = perform (default_action, args)
```

When command-line options call for multiple actions, those actions are merged by function `merge`. Options are processed left to right, and actions are merged in the same order. The initial action is always `DEFAULT`, which can appear only on the left. For most actions, the rightmost action takes precedence, but merging two `DUMP` actions performs them both.

S241d. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁S241c S241e▷

`merge: action * action -> action`

```
fun merge (_, a as FAIL _) = a
  | merge (a as FAIL _, _) = a
  | merge (DEFAULT, a) = a
  | merge (_, DEFAULT) = raise InternalError "DEFAULT on the right in MERGE"
  | merge (RUN_WITH _, right as RUN_WITH _) = right
  | merge (DUMP f, DUMP g) = DUMP (g o f)
  | merge (_, r) = FAIL "Interpret or dump, but don't try to do both"
```

Each possible command-line option is associated with an action. Options `-q` and `-qq` suppress prompts and echos. Options `-names` and `-primitives` dump information found in the initial basis.

S241e. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁S241d S242a▷

`actions : (string * action) list`

```
val actions =
  [ ("",    RUN_WITH (PROMPTING,     ECHOING))
  , ("-q",  RUN_WITH (NOT_PROMPTING, ECHOING))
  , ("-qq", RUN_WITH (NOT_PROMPTING, NOT_ECHOING))
  , ("-names",      DUMP (fn () => dump_names initialBasis))
  , ("-primitives", DUMP (fn () => dump_names primitiveBasis))
  , ("-help",       DUMP (fn () => !usage ()))
  ]
```

Now that the available command-line options are known, I can define a usage function. Function `CommandLine.name` returns the name by which the interpreter was invoked.

S242a. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁S241e S242b▷

```
val _ = usage := (fn () =>
  ( app eprint ["Usage:\n"]
  ; app (fn (option, action) =>
         app eprint ["       ", CommandLine.name (), " ", option, "\n"]) actions
  ))
```

Options are parsed by function `action`.

S242b. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁S242a S242c▷

`action : string -> action`

```
fun action option =
  case List.find (curry op = option o fst) actions
    of SOME (_, action) => action
     | NONE => FAIL ("Unknown option " ^ option)
```

A complete command-line is processed by computing the action associated with the command-line options, then performing that action with the remaining command-line arguments. Unless option `NORUN` is present in the `BPCOPTIONS` environment variable.

S242c. ⟨*look at command-line arguments, then run* S240c⟩+≡ (S379) ◁S242b

`strip_options : action -> string list -> action * string list`

```
fun strip_options a [] = (a, [])
  | strip_options a (arg :: args) =
      if String.isPrefix "-" arg andalso arg <> "-" then
          strip_options (merge (a, action arg)) args
      else
          (a, arg :: args)

val _ = if hasOption "NORUN" then ()
          else perform (strip_options DEFAULT (CommandLine.arguments ()))
```

H.10 Further reading

The `'a error` abstraction is an old functional-programming trick, first described by Spivey (1990). It has also been used to suppress spurious error messages in compilers (Ramsey 1999).

Appendix I contents

I Lexical analysis, parsing, and reading input using ML

How is a program represented? If you have worked through this book, you will believe (I hope) that the most fundamental and most useful representation of a program is its abstract-syntax tree. But syntax trees aren't easy to create or specify directly, so syntax usually has to be written using a sequence of characters. To help myself write parsers by hand, I have created[1] a set of higher-order functions designed especially to manipulate parsers. Such functions are known as *parsing combinators*. My parsing combinators appear in this appendix.

Most parsing techniques have been invented for use in compilers. and a typical compiler swallows programs in large gulps, one file at a time. Unlike these typical compilers, the interpreters in this book are interactive, and they swallow just one *line* at a time. Interactivity imposes additional requirements:

- Before reading a line of input, an interactive interpreter should issue a suitable *prompt*. The prompt should tell the user whether the parser is waiting for a new definition or is in the middle of parsing a current definition—which means that the line-reading functions must be in cahoots with the parser.

- If a parser encounters an error, it can't just give up. It needs to get itself back into a state where the user can continue to interact.

These requirements make my parsing combinators a bit different from standard ones. In particular, in order to be sure that the actions of printing a prompt and reading a line of input occur in the proper sequence, I manage these actions using the *lazy streams* defined in Section H.5.2. Unlike the lazy streams built into Haskell, these lazy streams can do input and output and can perform other actions.

Parsing is about turning a stream of lines (from a file or from a list of strings) into a stream of extended definitions. It happens in stages:

- In a stream of lines, each line is split into characters.

- A *lexical analyzer* turns a stream of characters into a stream of tokens. Using `streamConcatMap` with the lexical analyzer then turns a stream of lines into a stream of tokens.

- A *parser* turns a stream of tokens into a stream of syntax. I define parsers for expressions, true definitions, unit tests, and extended definitions.

The fundamental parser is `one`, which takes one token from a stream and produces that token. Other parsers are built on top of `one`, usually using higher-order functions. Functions `<$>` and `<*>` act like `map` for parsers, applying a function the result a parser returns. Function `sat` acts like `filter`, allowing a parser to fail if it doesn't

[1] I say "created," but it would be more accurate to say "stolen."

recognize its input. Functions <*>, <*, and *> combine parsers in sequence, and function <|> defines a parser as a choice between two other parsers. Functions `many` and `many1` turn a parser for a thing into a parser for a list of things; function `optional` does the same thing for ML's `option` type. These functions are known collectively as *parsing combinators*, and together they form a powerful language for defining lexical analyzers and parsers.

I divide parsers and parsing combinators into three groups:

- A *stream transformer* doesn't care what comes in or goes out; it is polymorphic in both the input and output type. Stream transformers are used to build both lexical analyzers and parsers.

- A *lexer* is a stream transformer that is specialized to take a stream of characters as input. Lexers may be defined with any output type, but a value of that output type should represent a token.

- A *parser* is a stream transformer that is specialized to take a stream of tokens as input. A parser's input stream also includes source-code locations and end-of-line markers. Parsers may be defined with any output type, but the rest of the interpreter is most interested in the parser that produces a stream of definitions (abstract-syntax trees).

The polymorphic functions are described in Table I.1 (page S248); the specialized functions are described in Table I.2 (page S255).

The code is divided among these chunks:

S246a. ⟨*common parsing code* S246a⟩≡
  ⟨*combinators and utilities for parsing located streams* S259b⟩
  ⟨*transformers for interchangeable brackets* S261c⟩
  ⟨*code used to debug parsers* S265c⟩
  ⟨*streams that issue two forms of prompts* S267a⟩

The functions defined in this appendix are useful for reading all kinds of input, not just computer programs, and I encourage you to use them in your own projects. But here are two words of caution: with so many abstractions in the mix, the parsers are tricky to debug. And while some parsers built from combinators are very efficient, mine aren't.

I.1 Stream transformers, which act as parsers

The purpose of a parser is to turn streams of input lines into streams of definitions. Intermediate representations may include streams of characters, tokens, types, expressions, and more. To handle all these different kinds of streams using a single set of operators, I define a type representing a *stream transformer*. A stream transformer from A to B takes a stream of A's as input and either succeeds, fails, or detects an error:

- If it succeeds, it consumes *zero or more* A's from the input stream and produces exactly one B. It returns a pair containing `OK` B plus whatever A's were not consumed.

- If it fails, it returns `NONE`.

- If it detects an error, it returns a pair containing `ERROR` m, where m is a message, plus whatever A's were not consumed.

A stream transformer from A to B has type `(`A`, `B`) transformer`.

S246b. ⟨*stream transformers and their combinators* S246b⟩≡ S247a ▷

```
type ('a, 'b) xformer =
  'a stream -> ('b error * 'a stream) option
```

`type ('a, 'b) xformer`

Applying `streamOfUnfold` (Section H.5.2) to an `('a, 'b) xformer` produces a function that maps a stream of A's to a stream of B's-with-error.

The stream-transformer abstraction supports many, many operations. These operations, known as *parsing combinators*, have been refined by functional programmers for over two decades, and they can be expressed in a variety of guises. The guise I have chosen uses notation from *applicative functors* and from the ParSec parsing library.

I begin very abstractly, by presenting combinators that don't actually consume any inputs. The next two sections present only "constant" transformers and "glue" functions that build transformers from other transformers. With those functions in place, I proceed to real, working parsing combinators. These combinators are split into two groups: "universal" combinators that work with any stream, and "parsing" combinators that expect a stream of tokens with source-code locations.

My design includes a lot of combinators. Too many, really. I would love to simplify the design, but simplifying software can be hard, and I don't want to delay the book by another year.

I.1.1 Error-free transformers and their composition

The `pure` combinator takes a value `y` of type B as argument. It returns an A-to-B transformer that consumes no A's as input and produces `y`.

S247a. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S246b S247b▷

```
fun pure y = fn xs => SOME (OK y, xs)
```

```
pure : 'b -> ('a, 'b) xformer
```

To build a stream transformer that reads inputs in sequence, I compose smaller stream transformers that read parts of the input. The sequential composition operator may look quite strange. To compose `tx_f` and `tx_b` in sequence, I use the infix operator `<*>`, which is pronounced "applied to." The composition is written `tx_f <*> tx_b`, and it works like this:

1. First `tx_f` reads some A's and produces a *function* `f` of type $B \to C$.
2. Next `tx_b` reads some more A's and produces a value `y` of type B.
3. The combination `tx_f <*> tx_b` reads no more input but simply applies `f` to `y` and returns `f y` (of type C) as its result.

This idea may seem crazy. How can reading a sequence of A's produce a function? The secret is that almost always, the function is produced by `pure`, without actually reading any A's, or it's the result of using the `<*>` operator to apply a Curried function to its first argument. But the read-and-produce-a-function idiom is a great way to do business, because when the parser is written using the `pure` and `<*>` combinators, the code resembles a Curried function application.

For the combination `tx_f <*> tx_b` to succeed, both `tx_f` and `tx_b` must succeed. Ensuring that two transformers succeed requires a nested case analysis.

`ERROR`	S221b
`type error`	S221b
`OK`	S221b
`type stream`	S228c

S247b. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S247a S249a▷

```
<*> : ('a, 'b -> 'c) xformer * ('a, 'b) xformer -> ('a, 'c) xformer
```

```
infix 3 <*>
fun tx_f <*> tx_b =
  fn xs => case tx_f xs
             of NONE => NONE
              | SOME (ERROR msg, xs) => SOME (ERROR msg, xs)
              | SOME (OK f, xs) =>
                  case tx_b xs
                    of NONE => NONE
                     | SOME (ERROR msg, xs) => SOME (ERROR msg, xs)
                     | SOME (OK y, xs) => SOME (OK (f y), xs)
```

Table I.1: Stream transformers and their combinators

Stream transformers; applying functions to transformers	
`type ('a, 'b) xformer`	
`pure`	`: 'b -> ('a, 'b) xformer`
`<*>`	`: ('a, 'b -> 'c) xformer * ('a, 'b) xformer -> ('a, 'c) xformer`
`<$>`	`: ('b -> 'c) * ('a, 'b) xformer -> ('a, 'c) xformer`
`<$>?`	`: ('b -> 'c option) * ('a, 'b) xformer -> ('a, 'c) xformer`
`<*>!`	`: ('a, 'b -> 'c error) xformer * ('a, 'b) xformer -> ('a, 'c) xformer`
`<$>!`	`: ('b -> 'c error) * ('a, 'b) xformer -> ('a, 'c) xformer`
Functions useful with <$> and <>*	
`fst`	`: ('a * 'b) -> 'a`
`snd`	`: ('a * 'b) -> 'b`
`pair`	`: 'a -> 'b -> 'a * 'b`
`curry`	`: ('a * 'b -> 'c) -> ('a -> 'b -> 'c)`
`curry3`	`: ('a * 'b * 'c -> 'd) -> ('a -> 'b -> 'c -> 'd)`
Combining transformers in sequence, alternation, or conjunction	
`<*`	`: ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'b) xformer`
`*>`	`: ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'c) xformer`
`<$`	`: 'b * ('a, 'c) xformer -> ('a, 'b) xformer`
`<|>`	`: ('a, 'b) xformer * ('a, 'b) xformer -> ('a, 'b) xformer`
`pzero`	`: ('a, 'b) xformer`
`anyParser`	`: ('a, 'b) xformer list -> ('a, 'b) xformer`
`<&>`	`: ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'c) xformer`
Transformers useful for both lexical analysis and parsing	
`one`	`: ('a, 'a) xformer`
`eos`	`: ('a, unit) xformer`
`sat`	`: ('b -> bool) -> ('a, 'b) xformer -> ('a, 'b) xformer`
`eqx`	`: ''b -> ('a, ''b) xformer -> ('a, ''b) xformer`
`notFollowedBy`	`: ('a, 'b) xformer -> ('a, unit) xformer`
`many`	`: ('a, 'b) xformer -> ('a, 'b list) xformer`
`many1`	`: ('a, 'b) xformer -> ('a, 'b list) xformer`
`optional`	`: ('a, 'b) xformer -> ('a, 'b option) xformer`
`peek`	`: ('a, 'b) xformer -> 'a stream -> 'b option`
`rewind`	`: ('a, 'b) xformer -> ('a, 'b) xformer`

The common case of creating tx_f using pure is normally written using the special operator <$>, which is also pronounced "applied to." It combines a B-to-C function with an A-to-B transformer to produce an A-to-C transformer.

S249a. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S247b S249c▷

```
<$> : ('b -> 'c) * ('a, 'b) xformer -> ('a, 'c) xformer
```

```
infixr 4 <$>
fun f <$> p = pure f <*> p
```

Functions that serve as f's are created in a variety of ways. Many such functions are Curried. Some of them are defined here.

S249b. ⟨*for working with curried functions:* id, fst, snd, pair, curry, *and* curry3 S249b⟩≡

```
fst    : ('a * 'b) -> 'a
snd    : ('a * 'b) -> 'b
pair   : 'a -> 'b -> 'a * 'b
curry  : ('a * 'b -> 'c) -> ('a -> 'b -> 'c)
curry3 : ('a * 'b * 'c -> 'd) -> ('a -> 'b -> 'c -> 'd)
```

```
fun id x = x
fun fst (x, y) = x
fun snd (x, y) = y
fun pair x y = (x, y)
fun curry  f x y   = f (x, y)
fun curry3 f x y z = f (x, y, z)
```

As an example, if a name is parsed by name and an expression is parsed by exp, then a name followed by an expression, such as might appear in a let binding, can be turned into $(name, expression)$ pair by the parser

```
pair <$> name <*> exp
```

(Parsing the actual μScheme syntax would also require a parser to handle the surrounding parentheses.) As another example, if a μScheme parser has seen a left bracket followed by the keyword if, it can call the parser

```
curry3 IFX <$> exp <*> exp <*> exp
```

which creates the abstract-syntax tree for an if expression.

The combinator <*> creates parsers that read things in sequence; but it can't make a choice. If any parser in the sequence fails, the whole sequence fails. A choice, as in "val or expression or define or use," is made by a choice operator. The choice operator is written <|> and pronounced "or." If t1 and t2 are both A-to-B transformers, then t1 <|> t2 is an A-to-B transformer that first tries t1, then tries t2. Transformer t1 <|> t2 succeeeds if either t1 or t2 succeeds, detects an error if either t1 or t2 detects an error, and fails only if both t1 and t2 fail. To assure that t1 <|> t2 has a predictable type no matter which transformer is chosen, both t1 and t2 have to have the same type.

S249c. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S249a S249d▷

```
<|> : ('a, 'b) xformer * ('a, 'b) xformer -> ('a, 'b) xformer
```

```
infix 1 <|>
fun t1 <|> t2 = (fn xs => case t1 xs of SOME y => SOME y | NONE => t2 xs)
```

I sometimes want to combine a list of parsers with the choice operator. I can do this by folding over the list, provided I have a "zero" parser, which always fails.

S249d. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S249c S249e▷

```
fun pzero _ = NONE
```

```
pzero : ('a, 'b) xformer
```

This parser obeys the algebraic law

$$\texttt{t <|> pzero} = \texttt{pzero <|> t} = \texttt{t}.$$

Because building choices from lists is common, I implement this special case as anyParser.

S249e. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S249d S250a▷

```
anyParser : ('a, 'b) xformer list -> ('a, 'b) xformer
```

```
fun anyParser ts = 
  foldr op <|> pzero ts
```

<*>	S247b
pure	S247a

I.1.2 Ignoring results produced by transformers

If a parser sees the stream of tokens

`(` `if` `(` `<` `x` `y` `)` `x` `y` `)`,

I want it to build an abstract-syntax tree using `IFX` and three expressions. The parentheses and keyword `if` serve to identify the `if`-expression and to make sure it is well formed, so the parser does have to read them from the input, but it doesn't need to do anything with the results that are produced. Using a parser and then ignoring the result is such a common operation that special abbreviations have evolved to support it.

The abbreviations are formed by modifying the `<*>` or `<$>` operator to remove the angle bracket on the side containing the result to be ignored. For example,

- Parser `p1 <* p2` reads the input of `p1` and then the input of `p2`, but it returns only the result of `p1`.
- Parser `p1 *> p2` reads the input of `p1` and then the input of `p2`, but it returns only the result of `p2`.
- Parser `v <$ p` parses the input the way `p` does, but it then ignores `p`'s result and instead produces the value `v`.

S250a. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S249e S250b▷

```
<*  : ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'b) xformer
 *> : ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'c) xformer
<$  : 'b                * ('a, 'c) xformer -> ('a, 'b) xformer
```

```
infix 6 <* *>
fun p1 <*  p2 = curry fst <$> p1 <*> p2
fun p1  *> p2 = curry snd <$> p1 <*> p2

infixr 4 <$
fun v <$ p = (fn _ => v) <$> p
```

I.1.3 At last, transformers that look at the input stream

None of the transformers above inspects an input stream. The fundamental operations are `pure`, `<*>`, and `<|>`; `pure` never looks at the input, and `<*>` and `<|>` simply sequence or alternate between other parsers which do the actual looking. Those parsers are up next.

The simplest input-inspecting parser is `one`. It's an A-to-A transformer that succeeds if and only if there is a value in the input. If there's no value in the input, `one` fails; it never signals an error.

S250b. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S250a S250c▷

```
one : ('a, 'a) xformer
```

```
fun one xs = case streamGet xs
               of NONE => NONE
                | SOME (x, xs) => SOME (OK x, xs)
```

The counterpart of `one` is a parser that succeeds if and only if there is *no* input—that is, if it is parsing the end of a stream. This parser, which is called `eos` ("end of stream"), can produce no useful result, so it produces the empty tuple, which has type `unit`.

S250c. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S250b S251a▷

```
eos : ('a, unit) xformer
```

```
fun eos xs = case streamGet xs
               of NONE => SOME (OK (), EOS)
                | SOME _ => NONE
```

Perhaps surprisingly, these are the only two standard parsers that inspect input. The only other parsing combinator that looks directly at input is the function `stripAndReportErrors`, which removes `ERROR` and `OK` from error streams.

It is sometimes useful to look at input without consuming it. For this purpose I define two functions: `peek` just looks at a transformed stream and maybe produces a value, whereas `rewind` changes any transformer into a transformer that behaves identically, but that doesn't consume any input. I use these functions either to debug, or to find the source-code location of the next token in a token stream.

S251a. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S250c S251b▷

```
peek : ('a, 'b) xformer -> 'a stream -> 'b option
```

```
  fun peek tx xs =
    case tx xs of SOME (OK y, _) => SOME y
                | _ => NONE
```

Given a transformer `tx`, transformer `rewind tx` computes the same value as `tx`, but when it's done, it rewinds the input stream back to where it was before it ran `tx`. The actions performed by `tx` can't be undone, but the inputs can be read again.

S251b. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S251a S251c▷

```
rewind : ('a, 'b) xformer -> ('a, 'b) xformer
```

```
  fun rewind tx xs =
    case tx xs of SOME (ey, _) => SOME (ey, xs)
                | NONE => NONE
```

I.1.4 Parsing combinators

Real parsers use `<$>`, `<*>`, `<|>`, and `one` as a foundation, then add ideas like these:

- Maybe the parser should succeed only if an input satisfies certain conditions. For example, if I want to parse numeric literals, I might want a character parser that succeeds only when the character is a digit.
- Most utterances in programming languages are made by composing things in sequence. For example, in μScheme, the characters in an identifier are a nonempty sequence of "ordinary" characters. And the arguments in a function application are a possibly empty sequence of expressions. Parser combinators for sequences are useful!
- Although I've avoided using "optional" syntax in the bridge languages, many, many programming languages do use constructs in which parts are optional. For example, in C, the use of an `else` clause with an `if` statement is optional. A parser combinator for this idiom can also be useful.

This section presents standard parsing combinators that help implement conditional parsers, parsers for sequences, and parsers for optional syntax.

`<$>`	S249a
`<*>`	S247b
`curry`	S249b
`EOS`	S228c
`fst`	S249b
`OK`	S221b
`snd`	S249b
`type stream`	S228c
`streamGet`	S229a

Parsers based on conditions

Combinator `sat` wraps an A-to-B transformer with a B-predicate such that the wrapped transformer succeeds only when the underlying transformer succeeds and produces a value that satisfies the predicate.

S251c. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S251b S252a▷

```
sat : ('b -> bool) -> ('a, 'b) xformer -> ('a, 'b) xformer
```

```
  fun sat p tx xs =
    case tx xs
      of answer as SOME (OK y, xs) => if p y then answer else NONE
       | answer => answer
```

Transformer `eqx b` is `sat` specialized to an equality predicate. It is typically used to recognize special characters like keywords and minus signs.

S252a. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S251c S252b▷

```
eqx : ''b -> ('a, ''b) xformer -> ('a, ''b) xformer
```

```
fun eqx y =
  sat (fn y' => y = y')
```

A predicate of type `('b -> bool)` asks, "Is this a thing?" But sometimes code wants to ask, "Is this a thing, and if so, what thing is it?" For example, a parser for Impcore or μScheme will want to know if an atom represents a numeric literal, but if so, it would also like to know *what* number is represented. Instead of a predicate, the parser would use a function of type `atom -> int option`. In general, an A-to-B transformer can be composed with a function of type B `->` C `option`, and the result is an A-to-C transformer. Because there's a close analogy with the application operator `<$>`, I notate the composition operator as `<$>?`, with a question mark.

S252b. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S252a S252c▷

```
<$>? : ('b -> 'c option) * ('a, 'b) xformer -> ('a, 'c) xformer
```

```
infixr 4 <$>?
fun f <$>? tx =
  fn xs => case tx xs
             of NONE => NONE
              | SOME (ERROR msg, xs) => SOME (ERROR msg, xs)
              | SOME (OK y, xs) =>
                  case f y
                    of NONE => NONE
                     | SOME z => SOME (OK z, xs)
```

Transformer `f <$>? tx` can be defined as `valOf <$> sat isSome (f <$> tx)`, but writing out the cases helps clarify what's going on.

A transformer might be run only if a another transformer succeeds on the same input. For example, the parser for μSmalltalk tries to parse an array literal only when it knows the input begins with a left bracket. Transformer `t1 <&> t2` succeeds only if both `t1` and `t2` succeed at the same point. An error in `t1` is treated as failure. The combined transformer looks at enough input to decide if `t1` succeeds, but it does not consume input consumed by `t1`—it consumes only the input of `t2`.

S252c. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S252b S252d▷

```
<&> : ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'c) xformer
```

```
infix 3 <&>
fun t1 <&> t2 = fn xs =>
  case t1 xs
    of SOME (OK _, _) => t2 xs
     | SOME (ERROR _, _) => NONE
     | NONE => NONE
```

A transformer can be complemented, turning success into failure and vice versa. Transformer `notFollowedBy t` succeeds if and only if `t` fails. Transformer `notFollowedBy t` may *look* at input, but it never *consumes* any input. This transformer is used when trying to read an integer literal, to make sure that the digits are not followed by a letter or other non-delimiting symbol.

S252d. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S252c S253a▷

```
notFollowedBy : ('a, 'b) xformer -> ('a, unit) xformer
```

```
fun notFollowedBy t xs =
  case t xs
    of NONE => SOME (OK (), xs)
     | SOME _ => NONE
```

Adding <&> and notFollowedBy to our library gives it the flavor of a little Boolean algebra for transformers: functions <&>, <|>, and notFollowedBy play the roles of "and," "or," and "not," and pzero plays the role of "false."

Transformers for sequences

Concrete syntax is full of sequences. A function takes a sequence of arguments, a program is a sequence of definitions, and a method definition contains a sequence of expressions. To create transformers that process sequences, I define functions many and many1. If t is an A-to-B transformer, then many t is an A-to-list-of-B transformer. It runs t as many times as possible. And even if t fails, many t always succeeds: when t fails, many t returns an empty list of B's.

S253a. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S252d S253b▷

```
fun many t =                  many  : ('a, 'b) xformer -> ('a, 'b list) xformer
  curry (op ::) <$> t <*> (fn xs => many t xs) <|> pure []
```

I'd really like to write that first alternative as

```
curry (op ::) <$> t <*> many t
```

but that formulation leads to instant death by infinite recursion. In your own parsers, it's a problem to watch out for.

Sometimes an empty list isn't acceptable. In such cases, I use many1 t, which succeeds only if t succeeds at least once—in which case it returns a nonempty list.

S253b. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S253a S253c▷

```
fun many1 t =                 many1 : ('a, 'b) xformer -> ('a, 'b list) xformer
  curry (op ::) <$> t <*> many t
```

Although many t always succeeds, many1 t can fail.

Both many and many1 are "greedy"; that is, they repeat t as many times as possible. Client code has to be careful to ensure that calls to many and many1 terminate. In particular, if t can succeed without consuming any input, then many t does not terminate. To pass many a transformer that succeeds without consuming input is therefor an *unchecked* run-time error. The same goes for many1.

Client code also has to be careful that when t sees something it doesn't recognize, it doesn't signal an error. In particular, t had better not be built with the <?> operator defined in chunk S260c below.

Sometimes instead of zero, one, or many B's, concrete syntax calls for zero or one; such a B might be called "optional." For example, a numeric literal begins with an optional minus sign. Function optional turns an A-to-B transformer into an A-to-optional-B transformer. Like many t, optional t always succeeds.

<$>	S249a
<*>	S247b
<|>	S249c
curry	S249b
ERROR	S221b
OK	S221b
pure	S247a
sat	S251c

S253c. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁S253b S254a▷

```
fun optional t =         optional : ('a, 'b) xformer -> ('a, 'b option) xformer
  SOME <$> t <|> pure NONE
```

Transformers made with many and optional succeed even when there is no input. They also succeed when there is input that they don't recognize.

I.1.5 Error-detecting transformers and their composition

Sometimes an error is detected not by a parser but by a function that is applied to the results of parsing. A classic example is a function definition: if the formal parameters are syntactically correct but contain a duplicate name, an error should be signaled. Formal parameters could be handled by a parser whose result type is name list error—but every transformer type already includes the possibility of error! I would prefer that the parser's result type be just `name list`, and that if duplicate names are detected, that the error be managed in the same way as a syntax error. To enable such management, I define `<*>!` and `<$>!` combinators, which merge function-detected errors with parser-detected errors.

S254a. ⟨*stream transformers and their combinators* S246b⟩+≡ ◁ S253c

```
<*>! : ('a, 'b -> 'c error) xformer * ('a, 'b) xformer -> ('a, 'c) xformer
<$>! : ('b -> 'c error)            * ('a, 'b) xformer -> ('a, 'c) xformer
```

```
infix 2 <*>!
fun tx_ef <*>! tx_x =
  fn xs => case (tx_ef <*> tx_x) xs
             of NONE => NONE
              | SOME (OK (OK y),      xs) => SOME (OK y,      xs)
              | SOME (OK (ERROR msg), xs) => SOME (ERROR msg, xs)
              | SOME (ERROR msg,      xs) => SOME (ERROR msg, xs)
infixr 4 <$>!
fun ef <$>! tx_x = pure ef <*>! tx_x
```

I.2 Lexical analyzers: transformers of characters

The interpreters in this book consume one line at a time. But characters *within* a line may be split into multiple *tokens*. For example, the line

```
(define list1 (x) (cons x '()))
```

should be split into the tokens

`(` `define` `list1` `(` `x` `)` `(` `cons` `x` `'` `(` `)` `)` `)`

This section defines reusable, specialized transformers that transform streams of characters into something else, usually tokens.

S254b. ⟨*support for lexical analysis* S254b⟩≡ S254c ▷

```
type 'a lexer = (char, 'a) xformer
```

```
type 'a lexer
```

The type `'a lexer` should be pronounced "lexer returning `'a`."

In popular languages, a character like a semicolon or comma usually does not join with other tokens to form a character. In this book, left and right brackets of all shapes keep to themselves and don't group with other characters. And in just about every non-esoteric language, blank space separates tokens. A character whose presence marks the end of one token (and possibly the beginning of the next) is called a *delimiter*. In this book, the main delimiter characters are whitespace and brackets. The other delimiter is the semicolon, which introduces a comment.

S254c. ⟨*support for lexical analysis* S254b⟩+≡ ◁ S254b S254d ▷

```
fun isDelim c =
  Char.isSpace c orelse Char.contains "()[]{};" c
```

```
isDelim : char -> bool
```

`Char.isSpace` recognizes all whitespace characters. `Char.contains` takes a string and a character and says if the string contains the character. These functions are in the initial basis of Standard ML.

All languages in this book ignore whitespace. Lexer `whitespace` is typically combined with another lexer using the `*>` operator.

S254d. ⟨*support for lexical analysis* S254b⟩+≡ ◁ S254c S256a ▷

```
val whitespace = many (sat Char.isSpace one)
```

```
whitespace : char list lexer
```

Table I.2: Transformers specialized for lexical analysis or parsing

Lexical analyzers; tokens

```
type 'a lexer = (char, 'a) xformer
isDelim       : char -> bool
whitespace    : char list lexer
intChars      : (char -> bool) -> char list lexer
intFromChars  : char list -> int error
intToken      : (char -> bool) -> int lexer
type token
tokenString   : token -> string
lexLineWith   : token lexer -> line -> token stream
```

Streams with end-of-line markers

```
type 'a eol_marked
drainLine     : 'a eol_marked stream -> 'a eol_marked stream
```

Parsers

```
type 'a parser = (token located eol_marked, 'a) xformer
eol           : ('a eol_marked, int) xformer
inline        : ('a eol_marked, 'a) xformer
token         : token parser
srcloc        : srcloc parser
noTokens      : unit parser
@@            : 'a parser -> 'a located parser
<?>           : 'a parser * string -> 'a parser
<!>           : 'a parser * string -> 'b parser
literal       : string -> unit parser
>--           : string * 'a parser -> 'a parser
--<           : 'a parser * string -> 'a parser
bracket       : string * string * 'a parser -> 'a parser
nodups        : string * string -> srcloc * name list
                                                -> name list error
safeTokens    : token located eol_marked stream -> token list
echoTagStream : line stream -> line stream
stripAndReportErrors
              : 'a error stream -> 'a stream
```

A complete, interactive source of abstract syntax

```
interactiveParsedStream : token lexer * 'a parser
                  -> string * line stream * prompts -> 'a stream
```

<*>	S247b
ERROR	S221b
type error	S221b
many	S253a
OK	S221b
one	S250b
pure	S247a
sat	S251c

Most languages in this book are, like Scheme, liberal about names. Just about any sequence of characters, as long as it is free of delimiters, can form a name. But there's one big exception: a sequence of digits forms an integer literal, not a name. Because integer literals introduce several complications, and because they are used in all the languages in this book, it makes sense to deal with the complications in one place: here.

Integer literals are subject to these rules:

- An integer literal may begin with a minus sign.
- It continues with one or more digits.
- If it is followed by character, that character must be a delimiter. (In other words, it must not be followed by a non-delimiter.)
- When the sequence of digits is converted to an `int`, the arithmetic used in the conversion must not overflow.

Function `intChars` does the lexical analysis to grab the characters; `intFromChars` handles the conversion and its potential overflow, and `intToken` puts everything together. Because not every language uses the same delimiters, both `intChars` and `intToken` receive a predicate that identifies delimiters.

S256a. ⟨*support for lexical analysis* S254b⟩+≡ ◁S254d S256b▷

```
fun intChars isDelim =          intChars : (char -> bool) -> char list lexer
  (curry (op ::) <$> eqx #"-" one <|> pure id) <*>
  many1 (sat Char.isDigit one) <*
  notFollowedBy (sat (not o isDelim) one)
```

Function `Char.isDigit`, like `Char.isSpace`, is part of Standard ML.

Function `intFromChars` composes three functions from Standard ML's initial basis. Function `implode` converts a list of characters to a string; `Int.fromString` converts a string to an `int option` (raising `Overflow` if the literal is too big); and `valOf` converts an `int option` to an `int`. The `Int.~` function, which is used when I see a minus sign, negates an integer. The `~` is meant to resemble a "high minus" sign, a notational convention that goes back at least to APL.

S256b. ⟨*support for lexical analysis* S254b⟩+≡ ◁S256a S256c▷

```
fun intFromChars (#"-" :: cs) =        intFromChars : char list -> int error
      intFromChars cs >>=+ Int.~
  | intFromChars cs =
      (OK o valOf o Int.fromString o implode) cs
      handle Overflow =>
        ERROR "this interpreter can't read arbitrarily large integers"
```

In this book, every language except μProlog can use `intToken`.

S256c. ⟨*support for lexical analysis* S254b⟩+≡ ◁S256b S256d▷

```
fun intToken isDelim =             intToken : (char -> bool) -> int lexer
  intFromChars <$>! intChars isDelim
```

All the bridge languages use balanced brackets, which may come in three shapes. So that lexers for different languages can share code related to brackets, bracket shapes and tokens are defined here.

S256d. ⟨*support for lexical analysis* S254b⟩+≡ ◁S256c S257a▷

```
datatype bracket_shape = ROUND | SQUARE | CURLY
```

Bracket tokens are added to a language-specific “pre-token” type by using the type constructor `plus_brackets`. Function `bracketLexer` takes as an argument a lexer for pre-tokens, and it returns a lexer for tokens:

S257a. ⟨*support for lexical analysis* S254b⟩+≡ ◁S256d S257b▷

```
type 'a plus_brackets
bracketLexer : 'a lexer -> 'a plus_brackets lexer
```

```
datatype 'a plus_brackets
  = LEFT  of bracket_shape
  | RIGHT of bracket_shape
  | PRETOKEN of 'a

fun bracketLexer pretoken
  =  LEFT  ROUND  <$ eqx #"(" one
 <|> LEFT  SQUARE <$ eqx #"[" one
 <|> LEFT  CURLY  <$ eqx #"{" one
 <|> RIGHT ROUND  <$ eqx #")" one
 <|> RIGHT SQUARE <$ eqx #"]" one
 <|> RIGHT CURLY  <$ eqx #"}" one
 <|> PRETOKEN <$> pretoken
```

For debugging and error messages, brackets and tokens can be converted to strings.

S257b. ⟨*support for lexical analysis* S254b⟩+≡ ◁S257a

```
plusBracketsString : ('a -> string) -> ('a plus_brackets -> string)
```

```
fun leftString ROUND  = "("
  | leftString SQUARE = "["
  | leftString CURLY  = "{"
fun rightString ROUND  = ")"
  | rightString SQUARE = "]"
  | rightString CURLY = "}"

fun plusBracketsString _   (LEFT shape)  = leftString shape
  | plusBracketsString _   (RIGHT shape) = rightString shape
  | plusBracketsString pts (PRETOKEN pt)  = pts pt
```

I.3 Parsers: reading tokens and source-code locations

To read definitions, expressions, and types, it helps to work at a higher level of abstraction than individual characters. All the parsers in this book use two stages: first a lexer groups characters into tokens, then a parser transforms tokens into syntax. Not all languages use the same tokens, so the code in this section assumes that the type `token` and function `tokenString` are defined. Function `tokenString` returns a string representation of any given token; it is used in debugging. As an example, the definitions used in μScheme appear in Appendix O (page S383).

Transforming a stream of characters to a stream of tokens to a stream of definitions should sound appealing, but it simplifies the story a little too much. That’s because if something goes wrong, a parser can't just throw up its hands. If an error occurs,

- The parser should say *where* things went wrong—at what *source-code location*.
- The parser should get rid of the bad tokens that caused the error.
- The parser should be able to keep going, without having to kill the interpreter and start over.

To support error reporting and recovery takes a lot of machinery. And that means a parser's input has to contain more than just tokens.

I.3.1 Flushing bad tokens

A standard parser for a batch compiler needs only to see a stream of tokens and to know from what source-code location each token came. A batch compiler can simply read all its input and report all the errors it wants to report.[2] But an interactive interpreter may not use an error as an excuse to read an indefinite amount of input. It must instead recover from the error and ready itself to read the next line. To do so, it needs to know where the line boundaries are! For example, if a parser finds an error on line 6, it should read all the tokens on line 6, throw them away, and start over again on line 7. And it should do this *without* reading line 7—reading line 7 will take an action and will likely have the side effect of printing a prompt. To mark line boundaries, I define a new type constructor `eol_marked`. A value of type `'a eol_marked` is either an end-of-line marker, or it contains a value of type `'a` that occurs in a line. A stream of such values can be drained up to the end of the line.

S258. ⟨*streams that track line boundaries* S258⟩≡ S259a ▷

```
type 'a eol_marked
drainLine : 'a eol_marked stream -> 'a eol_marked stream
```

```
datatype 'a eol_marked
  = EOL of int (* number of the line that ends here *)
  | INLINE of 'a

fun drainLine EOS = EOS
  | drainLine (SUSPENDED s)     = drainLine (demand s)
  | drainLine (EOL _    ::: xs) = xs
  | drainLine (INLINE _ ::: xs) = drainLine xs
```

[2] Batch compilers vary widely in the ambitions of their parsers. Some simple parsers report just one error and stop. Some sophisticated parsers analyze the entire input and report the smallest number of changes needed to make the input syntactically correct. And some ill-mannered parsers become confused after an error and start spraying meaningless error messages. But all of them have access to the entire input. The bridge-language interpreters don't.

To support a stream of marked lines—possibly marked, located lines—I define transformers `eol`, `inline`, and `srcloc`. The `eol` transformer returns the number of the line just ended.

S259a. ⟨*streams that track line boundaries* S258⟩+≡ ◁S258

```
eol      : ('a eol_marked, int) xformer
inline   : ('a eol_marked, 'a)  xformer
srcloc   : ('a located eol_marked, srcloc) xformer
```

```
local
  fun asEol (EOL n) = SOME n
    | asEol (INLINE _) = NONE
  fun asInline (INLINE x) = SOME x
    | asInline (EOL _)    = NONE
in
  fun eol    xs = (asEol    <$>? one) xs
  fun inline xs = (asInline <$>? many eol *> one) xs
  fun srcloc xs = rewind (fst <$> inline) xs
end
```

With source-code locations and end-of-line markers ready, I can now define true parsers: transformers that consume sequences of marked tokens.

I.3.2 Parsing located, in-line tokens

In each interpreter, a value of type `'a parser` is a transformer that takes a stream of located tokens set between end-of-line markers, and it returns a value of type `'a`, plus any leftover tokens. But each interpreter has its own token type, and the infrastructure needs to work with all of them. That is, it needs to be polymorphic. So a value of type `('t, 'a) polyparser` is a parser that takes tokens of some unknown type `'t`.

S259b. ⟨*combinators and utilities for parsing located streams* S259b⟩≡ (S246a) S259c ▷

```
type ('t, 'a) polyparser = ('t located eol_marked, 'a) xformer
```

When defining a parser, I want not to worry about the `EOL` and `INLINE` constructors. These constructors are essential for error recovery, but for parsing, they just get in the way. My first order of business is therefore to define analogs of `one` and `eos` that ignore `EOL`. Parser `token` takes one token; parser `srcloc` *looks* at the source-code location of a token, but leaves the token in the input; and parser `noTokens` succeeds only if there are no tokens left in the input. They are built on top of "utility" parsers `eol` and `inline`. The two utility parsers have different contracts; `eol` succeeds only when at `EOL`, but `inline` scans past `EOL` to look for `INLINE`.

S259c. ⟨*combinators and utilities for parsing located streams* S259b⟩+≡ (S246a) ◁S259b S260a ▷

```
token    : ('t, 't)   polyparser
noTokens : ('t, unit) polyparser
```

```
fun token    stream = (snd <$> inline)      stream
fun noTokens stream = (notFollowedBy token) stream
```

`:::`	S228c
`<$>`	S249a
`<$>?`	S252b
`demand`	S228b
`EOS`	S228c
`fst`	S249b
`many`	S253a
`notFollowedBy`	S252d
`one`	S250b
`rewind`	S251b
`snd`	S249b
`type stream`	S228c
`SUSPENDED`	S228c

Parser `noTokens` is not that same as `eos`: parser `eos` succeeds only when the input stream is empty, but `noTokens` can succeed when the input stream is *not* empty but contains only `EOL` markers—as is likely on the last line of an input file.

Source-code locations are useful by themselves, but they are also useful when paired with a result from a parser. For example, when parsing a message send for μSmalltalk, the source-code location of the send is used when writing a stack trace. To make it easy to add a source-code location to any result from any parser, I define the `@@` function. (Associate the word "at" with the idea of "location.") The code uses a dirty trick: it works because `srcloc` looks at the input but does not consume any tokens.

S260a. ⟨*combinators and utilities for parsing located streams* S259b⟩+≡ (S246a) ◁S259c S260b▷

```
                        @@ : ('t, 'a) polyparser -> ('t, 'a located) polyparser
fun @@ p = pair <$> srcloc <*> p
```

I usually want names to contain only printing ASCII characters. Imagine a student who thinks they've written a minus sign but has actually written some accursed Unicode character that looks just like a minus sign. Nobody is more confused or frustrated, and justifiably so.

S260b. ⟨*combinators and utilities for parsing located streams* S259b⟩+≡ (S246a) ◁S260a S260c▷

```
                    asAscii : ('t, string) polyparser -> ('t, string) polyparser
fun asAscii p =
  let fun good c = Char.isPrint c andalso Char.isAscii c
      fun warn (loc, s) =
        case List.find (not o good) (explode s)
          of NONE => OK s
           | SOME c =>
               let val msg =
                     String.concat ["name \"", s, "\" contains the ",
                                    "non-ASCII or non-printing byte \"",
                                    Char.toCString c, "\""]
               in  synerrorAt msg loc
               end
  in  warn <$>! @@ p
  end
```

I.3.3 *Parsers that report errors*

A typical syntactic form (expression, unit test, or definition, for example) is parsed by a sequence of alternatives separated with `<|>`. When no alternative succeeds, the collective should usually be reported as a syntax error. An error-reporting parser can be created using the `<?>` function: parser `p <?> what` succeeds when `p` succeeds, but when `p` fails, parser `p <?> what` reports an error: it expected `what`. The error says what the parser was expecting, and it gives the source-code location of the unrecognized token. If there is no token, there is no error—at end of file, rather than signal an error, a parser made using `<?>` fails. An example appears in the parser for extended definitions in μScheme (chunk S389a).

S260c. ⟨*combinators and utilities for parsing located streams* S259b⟩+≡ (S246a) ◁S260b S261a▷

```
infix 0 <?>            <?> : ('t, 'a) polyparser * string -> ('t, 'a) polyparser
fun p <?> what = p <|> synerrorAt ("expected " ^ what) <$>! srcloc
```

The `<?>` operator must not be used to define a parser that is passed to `many`, `many1`, or `optional` In that context, if parser `p` fails, it must not signal an error; it must instead propagate the failure to `many`, `many1`, or `optional`, so those combinators know there is not a `p` there.

Another common error-detecting technique is to use a parser `p` to detect some input that shouldn't be there. For example, a parser is just starting to read a definition, the input shouldn't begin with a right parenthesis. I can write a parser `p` that recognizes a right parenthesis, but I can't simply combine `p` with `synerrorAt` and `srcloc` in the same way that `<?>` does, because I want my combined parser to do two things: *consume* the tokens recognized by `p`, and also *report* the error at the location of the first of those tokens. I can't use `synerrorAt` until *after* `p` succeeds, but I have to use `srcloc` on the input stream as it is *before* `p` is run. I solve this problem by defining a special combinator that keeps a copy of the tokens inspected by p. If parser p succeeds, then parser p <!> msg consumes the tokens consumed by p and reports error msg at the location of p's first token.

S261a. ⟨*combinators and utilities for parsing located streams* S259b⟩+≡ (S246a) ◁S260c S265a▷

```
infix 4 <!>                 <!> : ('t, 'a) polyparser * string -> ('t, 'b) polyparser
fun p <!> msg =
  fn tokens => (case p tokens
                  of SOME (OK _, unread) =>
                       let val outcome =
                             case peek srcloc tokens
                              of SOME loc => synerrorAt msg loc
                               | NONE => ERROR msg

                       in  SOME (outcome, unread)
                       end
                   | _ => NONE)
```

Function <!> is adequate for simple cases, but to produce a really good error message, I might wish to use the result from p to build a message. My interpreters produce such messages only for text appearing in brackets, so errorAtEnd triggers only when p parses tokens that are followed by a right bracket.

S261b. ⟨*definition of function* errorAtEnd S261b⟩≡ (S262c)

```
errorAtEnd : ('t plus_brackets, 'a) polyparser * ('a -> string list) -> ('t plus_brackets, 'b) polyparser
```

```
infix 4 errorAtEnd
fun p errorAtEnd mkMsg =
  fn tokens =>
    (case (p <* rewind right) tokens
       of SOME (OK s, unread) =>
            let val outcome =
                  case peek srcloc tokens
                    of SOME loc => synerrorAt ((concat o mkMsg) s) loc
                     | NONE => ERROR ((concat o mkMsg) s)
            in  SOME (outcome, unread)
            end
        | _ => NONE)
```

I.3.4 *Parsers that involve brackets*

Almost every language in this book uses a parenthesis-prefix syntax (Scheme syntax) in which round and square brackets must match, but are otherwise interchangeable.[3] Brackets are treated specially by the plus_brackets type (page S257), which identifies every token as a left bracket, a right bracket, or a "pre-token." Each of these alternatives is supported by its own parser. A parser that finds a bracket returns the bracket's shape and location; a parser the finds a pre-token returns the pre-token.

S261c. ⟨*transformers for interchangeable brackets* S261c⟩≡ (S246a) S262a▷

```
left  : ('t plus_brackets, bracket_shape located) polyparser
right : ('t plus_brackets, bracket_shape located) polyparser
pretoken : ('t plus_brackets, 't) polyparser
```

```
fun left  tokens = ((fn (loc, LEFT  s) => SOME (loc, s) | _ => NONE) <$>? inline) tokens
fun right tokens = ((fn (loc, RIGHT s) => SOME (loc, s) | _ => NONE) <$>? inline) tokens
fun pretoken stream = ((fn PRETOKEN t => SOME t | _ => NONE) <$>? token) stream
```

[3] I have spent entirely too much time working with Englishmen who call parentheses "brackets." I now find it hard even to *say* the word "parenthesis," let alone type it.

Every interpreter needs to be able to complain when it encounters an unexpected right bracket. An interpreter can build a suitable parser by passing a message to `badRight`. Since the parser never succeeds, it can have any result type.

S262a. ⟨*transformers for interchangeable brackets* S261c⟩+≡ (S246a) ◁S261c S262b▷

```
fun badRight msg =   badRight : string -> ('t plus_brackets, 'a) polyparser
  (fn (loc, shape) => synerrorAt (msg ^ " " ^ rightString shape) loc) <$>! right
```

In this book, round and square brackets are interchangeable, but curly brackets are special. Predicate `notCurly` identifies those non-curly, interchangeable bracket shapes. Parser `leftCurly` is just like `left`, except it recognizes only curly left brackets.

S262b. ⟨*transformers for interchangeable brackets* S261c⟩+≡ (S246a) ◁S262a S262c▷

```
                notCurly  : bracket_shape located -> bool
                leftCurly : ('t plus_brackets, bracket_shape located) polyparser
fun notCurly (_, CURLY) = false
  | notCurly _          = true
fun leftCurly tokens = sat (not o notCurly) left tokens
```

With the bracket parsers defined, I can use Noweb to drop the definition of function `errorAtEnd` into this spot.

S262c. ⟨*transformers for interchangeable brackets* S261c⟩+≡ (S246a) ◁S262b S262d▷
⟨*definition of function* `errorAtEnd` S261b⟩

Brackets by themselves are all very well, but what I really want is to parse syntax that is wrapped in *matching* brackets. But what if something goes wrong inside the brackets? In that case, I want each of my parsers to skip tokens until it gets to the matching right bracket, and I'll likely want it to report the source-code location of the *left* bracket. To look ahead for a right bracket is the job of parser `matchingRight`. This parser is called when a left bracket has already been consumed, and it searches the input stream for a right bracket, skipping every left/right pair that it finds in the interim. Because it's meant for error handling, it always succeeds. And to communicate its findings, it produces one of three outcomes:

- Result `FOUND_RIGHT (`*loc*`, `*s*`)` says, "I found a right bracket exactly where I expected to, and its shape and location are *s* and *loc*."
- Result `SCANNED_TO_RIGHT` *loc* says, "I didn't find a right bracket at *loc*, but I scanned to a matching right bracket eventually."
- Result `NO_RIGHT` says, "I scanned the entire input without finding a matching right bracket."

This result is defined as follows:

S262d. ⟨*transformers for interchangeable brackets* S261c⟩+≡ (S246a) ◁S262c S263a▷

```
datatype right_result                                          type right_result
  = FOUND_RIGHT      of bracket_shape located
  | SCANNED_TO_RIGHT of srcloc  (* location where scanning started *)
  | NO_RIGHT
```

A value of type `right_result` is produced by parser `matchingRight`. A right bracket in the expected position is successfully found by the `right` parser; when tokens have to be skipped, they are skipped by parser scanToClose. The “matching” is done purely by counting left and right brackets; scanToClose does not look at shapes.

S263a. ⟨*transformers for interchangeable brackets* S261c⟩+≡ (S246a) ◁S262d S263b▷

`matchingRight : ('t, right_result) pb_parser`

```
type ('t, 'a) pb_parser = ('t plus_brackets, 'a) polyparser
fun matchingRight tokens =
  let fun scanToClose tokens =
        let val loc = getOpt (peek srcloc tokens, ("end of stream", 9999))
            fun scan nlp tokens =
              (* nlp is the number of unmatched left parentheses *)
              case tokens
                of EOL _                  ::: tokens => scan nlp tokens
                 | INLINE (_, PRETOKEN _) ::: tokens => scan nlp tokens
                 | INLINE (_, LEFT  _)    ::: tokens => scan (nlp+1) tokens
                 | INLINE (_, RIGHT _)    ::: tokens =>
                     if nlp = 0 then
                       pure (SCANNED_TO_RIGHT loc) tokens
                     else
                       scan (nlp-1) tokens
                 | EOS         => pure NO_RIGHT tokens
                 | SUSPENDED s => scan nlp (demand s)
        in  scan 0 tokens
        end
  in  (FOUND_RIGHT <$> right <|> scanToClose) tokens
  end
```

The right-bracket result is used in function matchBrackets, which looks for a right bracket after a thing is matched. Function matchBrackets *does* look at shapes: if the right bracket is immediately present (FOUND_RIGHT) and is of the proper shape, then the match succeeds. Otherwise, it produces a suitable error.

S263b. ⟨*transformers for interchangeable brackets* S261c⟩+≡ (S246a) ◁S263a S264a▷

`matchBrackets : string -> bracket_shape located -> 'a -> right_result -> 'a error`

```
fun matchBrackets _ (loc, left) a (FOUND_RIGHT (loc', right)) =
      if left = right then
        OK a
      else
        synerrorAt (rightString right ^ " does not match " ^ leftString left ^
                 (if loc <> loc' then " at " ^ srclocString loc else "")) loc'
  | matchBrackets _ (loc, left) _ NO_RIGHT =
      synerrorAt ("unmatched " ^ leftString left) loc
  | matchBrackets e (loc, left) _ (SCANNED_TO_RIGHT loc') =
      synerrorAt ("expected " ^ e) loc
```

The bracket matcher is then used to help wrap other parsers in brackets. A parser may be wrapped in a variety of ways, depending on what may be allowed to fail without causing an error.

- To wrap parser p in matching round or square brackets when p may fail: use liberalBracket. If p succeeds but the brackets don’t match, that’s an error.
- To wrap parser p in matching round or square brackets when p must succeed: use bracket.
- To wrap parser p in matching curly brackets when p must succeed: use curlyBracket.

- To put parser `p` after a keyword, all wrapped in brackets: use `bracketKeyword`. Once the keyword is seen, `p` must not fail—if it does, that's an error.

Each of these functions takes a parameter `expected` of type `string`; when anything goes wrong, this parameter says what the parser was expecting.

S264a. ⟨*transformers for interchangeable brackets* S261c⟩+≡ (S246a) ◁S263b S264b▷

```
liberalBracket : string * ('t, 'a) pb_parser -> ('t, 'a) pb_parser
bracket        : string * ('t, 'a) pb_parser -> ('t, 'a) pb_parser
curlyBracket   : string * ('t, 'a) pb_parser -> ('t, 'a) pb_parser
bracketKeyword : ('t, 'keyword) pb_parser * string * ('t, 'a) pb_parser ->
                                                          ('t, 'a) pb_parser
```

```
fun liberalBracket (expected, p) =
  matchBrackets expected <$> sat notCurly left <*> p <*>! matchingRight
fun bracket (expected, p) =
  liberalBracket (expected, p <?> expected)
fun curlyBracket (expected, p) =
  matchBrackets expected <$> leftCurly <*> (p <?> expected) <*>! matchingRight
fun bracketKeyword (keyword, expected, p) =
  liberalBracket (expected, keyword *> (p <?> expected))
```

The `bracketKeyword` function is what's used to build parsers for `if`, `lambda`, and many other syntactic forms. And if one of these parsers fails, I want it to show the programmer what's expected, like for example `"(if e1 e2 e3)"`. The expectation is represented by a *usage string*. A usage string begins with a left bracket, which is followed by its keyword. I want not to write the keyword twice, so `usageParser` pulls the keyword out of the usage string—using a parser.

S264b. ⟨*transformers for interchangeable brackets* S261c⟩+≡ (S246a) ◁S264a

```
usageParser : (string -> ('t, string) pb_parser) ->
                    string * ('t, 'a) pb_parser -> ('t, 'a) pb_parser
```

```
fun usageParser keyword =
  let val left = eqx #"(" one <|> eqx #"[" one
      val getkeyword = left *> (implode <$> many1 (sat (not o isDelim) one))
  in  fn (usage, p) =>
        case getkeyword (streamOfList (explode usage))
          of SOME (OK k, _) => bracketKeyword (keyword k, usage, p)
           | _ => raise InternalError ("malformed usage string: " ^ usage)
  end
```

I.3.5 Detection of duplicate names

Most of the languages in this book allow you to define functions or methods that take formal parameters. It is never permissible to use the same name for formal parameters in two different positions. There are surprisingly many other places where it's not acceptable to have duplicates in a list of strings. Function `nodups` takes two Curried arguments: a pair saying what kind of thing might be duplicated and where it appeared, followed by a pair containing a list of names and the source-code location of the list. If there are no duplicates, it returns `OK` applied to the list of names; otherwise it returns an `ERROR`. Function `nodups` is typically applied to a pair of strings, after which the result is applied to a parser using the `<$>!` function.

S265a. ⟨*combinators and utilities for parsing located streams* S259b⟩+≡ (S246a) ◁S261a S265b▷

```
nodups : string * string -> srcloc * name list -> name list error
```

```
fun nodups (what, context) (loc, names) =
  let fun dup [] = OK names
        | dup (x::xs) =
            if List.exists (fn y => y = x) xs then
              synerrorAt (what ^ " " ^ x ^ " appears twice in " ^ context) loc
            else
              dup xs
  in  dup names
  end
```

Function List.exists is like the μScheme exists?. It is in the initial basis for Standard ML.

I.3.6 Detection of reserved words

To rule out such nonsense as "(val if 3)," parsers use function rejectReserved, which issues a syntax-error message if a name is on a list of reserved words.

S265b. ⟨*combinators and utilities for parsing located streams* S259b⟩+≡ (S246a) ◁S265a

```
rejectReserved : name list -> name -> name error
```

```
fun rejectReserved reserved x =
  if member x reserved then
    ERROR ("syntax error: " ^ x ^ " is a reserved word and " ^
           "may not be used to name a variable or function")
  else
    OK x
```

I.3.7 Code used to debug parsers

When debugging parsers, I often find it helpful to dump out the tokens that a parser is looking at. I want to dump only the tokens that are available *without* triggering the action of reading another line of input. To get those tokens, function safeTokens reads until it has got to *both* an end-of-line marker *and* a suspension whose value has not yet been demanded.

S265c. ⟨*code used to debug parsers* S265c⟩≡ (S246a) S266a▷

```
safeTokens : 'a located eol_marked stream -> 'a list
```

```
fun safeTokens stream =
  let fun tokens (seenEol, seenSuspended) =
            let fun get (EOL _        ::: ts) = if seenSuspended then []
                                                 else tokens (true, false) ts
                  | get (INLINE (_, t) ::: ts) = t :: get ts
                  | get  EOS                   = []
                  | get (SUSPENDED (ref (PRODUCED ts))) = get ts
                  | get (SUSPENDED s) = if seenEol then []
                                        else tokens (false, true) (demand s)
            in   get
            end
  in  tokens (false, false) stream
  end
```

Another way to debug is to show whatever input tokens might cause an error. They can be shown using function `showErrorInput`, which transforms an ordinary parser into a parser that, when it errors, shows the input that caused the error. It should be applied routinely to every parser you build.

S266a. ⟨*code used to debug parsers* S265c⟩+≡ (S246a) ◁S265c S266b▷

```
showErrorInput : ('t -> string) -> ('t, 'a) polyparser -> ('t, 'a) polyparser
```

```
fun showErrorInput asString p tokens =
  case p tokens
    of result as SOME (ERROR msg, rest) =>
         if String.isSubstring " [input: " msg then
           result
         else
           SOME (ERROR (msg ^ " [input: " ^
                        spaceSep (map asString (safeTokens tokens)) ^ "]"),
               rest)
     | result => result
```

What if a parser doesn't cause an error, but it fails when you were expecting it to succeed? Try applying `wrapAround` to it; using `wrapAround` with a parser `p` shows what `p` was looking for, what tokens it was looking at, and whether it found something.

S266b. ⟨*code used to debug parsers* S265c⟩+≡ (S246a) ◁S266a

```
wrapAround : ('t -> string) -> string -> ('t, 'a) polyparser -> ('t, 'a) polyparser
```

```
fun wrapAround tokenString what p tokens =
  let fun t tok = " " ^ tokenString tok
      val _ = app eprint ["Looking for ", what, " at"]
      val _ = app (eprint o t) (safeTokens tokens)
      val _ = eprint "\n"
      val answer = p tokens
      val _ = app eprint [ case answer of NONE => "Didn't find "
                                        | SOME _ => "Found "
                         , what, "\n"
                         ]
  in  answer
  end handle e =>
        ( app eprint ["Search for ", what, " raised ", exnName e, "\n"]
        ; raise e
        )
```

I.4 Streams that lex, parse, and prompt

The preceding sections of this appendix show individual pieces of machinery that are useful for making parsers. This final section puts those pieces together. The functions shown here take a stream of input lines, a lexer, and a parser, and they produce a stream of high-level syntactic objects like definitions. With prompts! The functions in this section determine the prompts, handle errors when they occur, and even copy special tagged lines that are used to test the examples in this book.

Support for testing

I begin with testing support. As in the C code, I want each interpreter to print out any line read that begins with the special string `;#`. This string is a formal comment that helps test chunks marked ⟨*transcript*⟩. The strings are printed in a modular way: a post-stream action prints any line meeting the criterion. Function echoTagStream transforms a stream of lines to a stream of lines, adding the behavior I want.

S267a. ⟨*streams that issue two forms of prompts* S267a⟩≡ (S246a) S267b ▷

```
echoTagStream : line stream -> line stream
```

```
fun echoTagStream lines =
  let fun echoIfTagged line =
        if (String.substring (line, 0, 2) = ";#" handle _ => false) then
          print line
        else
          ()
  in  postStream (lines, echoIfTagged)
  end
```

Issuing messages for error values

Next is error handling. A process that can detect errors produces a stream of type 'a error stream, for some unspecified type 'a. The ERROR and OK tags can be removed by reporting errors and passing on values tagged OK, resulting in a new stream of type 'a stream. Values tagged with OK are passed on to the output stream unchanged; messages tagged with ERROR are printed to standard error, using eprintln.

S267b. ⟨*streams that issue two forms of prompts* S267a⟩+≡ (S246a) ◁ S267a S267c ▷

```
stripAndReportErrors : 'a error stream -> 'a stream
```

```
fun stripAndReportErrors xs =
  let fun next xs =
        case streamGet xs
          of SOME (ERROR msg, xs) => (eprintln msg; next xs)
           | SOME (OK x, xs) => SOME (x, xs)
           | NONE => NONE
  in  streamOfUnfold next xs
  end
```

Using stripAndReportErrors, I can turn a lexical analyzer into a function that takes an input line and returns a stream of tokens. Any errors detected during lexical analysis are printed without any information about source-code locations. That's because, to keep things somewhat simple, I've chosen to do lexical analysis on one line at a time, and my code doesn't keep track of the line's source-code location.

S267c. ⟨*streams that issue two forms of prompts* S267a⟩+≡ (S246a) ◁ S267b S267d ▷

```
lexLineWith : 't lexer -> line -> 't stream
```

```
fun lexLineWith lexer =
  stripAndReportErrors o streamOfUnfold lexer o streamOfList o explode
```

When an error occurs during parsing, I want the parser to drain the rest of the tokens on the line where the error occurred. And errors aren't stripped yet; errors are passed on to the interactive stream because when an error is detected, the prompt may need to be changed.

S267d. ⟨*streams that issue two forms of prompts* S267a⟩+≡ (S246a) ◁ S267c S268 ▷

```
parseWithErrors : ('t, 'a) polyparser ->
                    't located eol_marked stream -> 'a error stream
```

```
fun parseWithErrors parser =
  let fun adjust (SOME (ERROR msg, tokens)) =
            SOME (ERROR msg, drainLine tokens)
        | adjust other = other
  in  streamOfUnfold (adjust o parser)
  end
```

drainLine	S258
eprint	S215b
eprintln	S215b
ERROR	S221b
type error	S221b
OK	S221b
type polyparser	S259b
postStream	S231b
safeTokens	S265c
spaceSep	S214e
type stream	S228c
streamGet	S229a
streamOfList	S229b
streamOfUnfold	S230b

Prompts

Each interpreter in this book issues prompts using the model established by the Unix shell. This model uses two prompt strings. The first prompt string, called `ps1`, is issued when starting to read a definition. The second prompt string, called `ps2`, is issued when in the middle of reading a definition. Prompting can be disabled by making both `ps1` and `ps2` empty.

S268. ⟨*streams that issue two forms of prompts* S267a⟩+≡ (S246a) ◁S267d S269▷

```
type prompts   = { ps1 : string, ps2 : string }
val stdPrompts = { ps1 = "-> ", ps2 = "   " }
val noPrompts  = { ps1 = "", ps2 = "" }
```

```
type prompts
stdPrompts : prompts
noPrompts  : prompts
```

Building a reader

All that is left is to combine lexer and parser. The combination manages the flow of information from the input through the lexer and parser, and by monitoring the flow of tokens in and syntax out, it arranges that the right prompts (`ps1` and `ps2`) are printed at the right times. The flow of information involves multiple steps:

1. The input is a stream of lines. The stream is transformed with `preStream` and `echoTagStream`, so that a prompt is printed before every line, and when a line contains the special tag, that line is echoed to the output.

2. Each line is converted to a stream of tokens by function `lexLineWith lexer`. Each token is then paired with a source-code location and, tagged with `INLINE`, and the stream of tokens is followed by an `EOL` value. This extra decoration transforms the `token stream` provided by the lexer to the `token located eol_marked stream` needed by the parser. The work is done by function `lexAndDecorate`, which needs a *located* line.

   The moment a token is successfully taken from the stream, a `postStream` action sets the prompt to `ps2`.

3. A final stream of definitions is computed by composing `locatedStream` to add source-code locations, `streamConcatMap lexAndDecorate` to add decorations, and `parseWithErrors parser` to parse. The entire composition is applied to the stream of lines created in step 1.

The composition is orchestrated by function `interactiveParsedStream`.

To deliver the right prompt in the right situation, `interactiveParsedStream` stores the current prompt in a mutable cell called `thePrompt`. The prompt is initially `ps1`, and it stays `ps1` until a token is delivered, at which point the `postStream` action sets it to `ps2`. But every time a new definition is demanded, a `preStream` action on the syntax stream `xdefs_with_errors` resets the prompt to `ps1`. This combination of pre- and post-stream actions, on different streams, ensures that the prompt is always appropriate to the state of the parser.

S269. ⟨*streams that issue two forms of prompts* S267a⟩+≡ (S246a) ◁S268

```
interactiveParsedStream : 't lexer * ('t, 'a) polyparser ->
                          string * line stream * prompts -> 'a stream
lexAndDecorate : srcloc * line -> 't located eol_marked stream
```

```
fun ('t, 'a) interactiveParsedStream (lexer, parser) (name, lines, prompts) =
  let val { ps1, ps2 } = prompts
      val thePrompt = ref ps1
      fun setPrompt ps = fn _ => thePrompt := ps

      val lines = preStream (fn () => print (!thePrompt), echoTagStream lines)

      fun lexAndDecorate (loc, line) =
        let val tokens = postStream (lexLineWith lexer line, setPrompt ps2)
        in  streamMap INLINE (streamZip (streamRepeat loc, tokens)) @@@
            streamOfList [EOL (snd loc)]
        end

      val xdefs_with_errors : 'a error stream =
        (parseWithErrors parser o streamConcatMap lexAndDecorate o locatedStream)
        (name, lines)
  in
      stripAndReportErrors (preStream (setPrompt ps1, xdefs_with_errors))
  end
```

I.5 Further reading

You could easily spend three months studying nothing but parsing.

If you've heard of tools like Yacc, Bison, or Elkhound, they are all based on "LR parsing," which was invented by Knuth (1965). Knuth's paper is really nice.

The theory of parsing (as of the early 1970s) was elucidated in great detail by Aho and Ullman (1972). This book represents a big chunk of the work for which Aho and Ullman received the Turing Award.

There is still a place for hand-written recursive-descent parsers, and nobody is better at them than Wirth (1977).

The parsing functions in this book are inspired by *parsing combinators*. These combinators are nicely introduced in a tutorial by Hutton and Meijer (1996) The notation of the <$> and <*> functions is suggested in a great paper by McBride and Paterson (2008).

The `streamOfUnfold` function, which is used to help pull everything together, is inspired by Gibbons and Jones (1998).

The `ERROR` values and `'a error` type are drawn from my own research (Ramsey 1999).

Appendix J contents

J

Supporting discriminated unions in C

This appendix presents an ML program that reads the data descriptions from Chapters 1 to 4 and produces C declarations of types that represent the data and C functions that operate on the data. The data descriptions are written in μASDL, a domain-specific language that is inspired the Zephyr Abstract Syntax Description Language (Wang et al. 1997). A description might look like this:

S271. ⟨*example input* S271⟩≡

```
Lambda  = (Namelist formals, Exp body)
Def*    = VAL     (Name name, Exp exp)
        | EXP     (Exp)
        | DEFINE  (Name name, Lambda lambda)
        | USE     (Name)
```

For a name like `Lambda`, which defines a product (record), the program produces declarations like these:

```
typedef struct Lambda Lambda;
struct Lambda { Namelist formals; Exp body; };
Lambda mkLambda(Namelist formals, Exp body);
```

For a name like `Def`, which defines a sum, C code needs to identify which alternative of the sum is meant. This program creates a type `Defalt`, which identifies an alternative, as well as other declarations related to `Def`:

```
typedef struct Def *Def;
typedef enum { VAL, EXP, DEFINE, USE } Defalt;

Def mkVal(Name name, Exp exp);
Def mkExp(Exp exp);
Def mkDefine(Name name, Lambda lambda);
Def mkUse(Name use);

struct Def {
    Defalt alt;
    union {
        struct { Name name; Exp exp; } val;
        Exp exp;
        struct { Name name; Lambda lambda; } define;
        Name use;
    };
};
```

μASDL is implemented in Standard ML, so it uses the infrastructure described in Chapter 5 and in Appendices H and I. The code is structured as follows:

S272a. ⟨*asdl.sml* S272a⟩≡

```
⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization (from chunk 697b)⟩
⟨abstract syntax for μASDL S274a⟩
⟨lexical analysis for μASDL S272b⟩
⟨parsers for μASDL S273b⟩
⟨prettyprinting combinators S277a⟩
⟨C types S277b⟩
⟨prettyprinting C types S278b⟩
⟨converting sums and products to C types S279c⟩
⟨functions that build documents to be emitted S283a⟩
⟨function process, which reads input and writes output S284c⟩
val defstream = interactiveParsedStream (asdlToken, def <?> "definition")
val defs = defstream ("standard input", filelines TextIO.stdIn, noPrompts)

val usage =
  concat ["Usage: ", CommandLine.name(), " cfile nwfile name language"]

val _ = case CommandLine.arguments ()
          of [c, web, name, lang] => process c web name lang defs
           | [base, name, lang] => (* legacy usage *)
               process (base ^ "-code.c") (base ^ ".xnw") name lang defs
           | _ => eprintln usage
```

J.1 Lexical analysis

μASDL reserves symbols `*`, `=`, `|`, comma, and brackets. A token in all upper case is a constructor, and anything else is a name. Constructors, value constructors in μML or Standard ML, identify the alternatives in a sum type. Type `token` is defined by adding brackets (Appendix I) to the three “pre-tokens.”

S272b. ⟨*lexical analysis for μASDL* S272b⟩≡ (S272) S273a ▷

```
datatype pretoken
  = RESERVED of char
  | CONSTR   of name (* constructor *)
  | NAME     of name
type token = pretoken plus_brackets
```

A `pretoken` can be converted to a string.

S272c. ⟨*definitions of type* `token` *and function* `tokenString` *for μASDL* S272c⟩≡

`pretokenString : pretoken -> string`

```
⟨lexical analysis for μASDL S272b⟩
fun pretokenString (RESERVED c) = str c
  | pretokenString (NAME n)     = n
  | pretokenString (CONSTR c)   = c
```

The lexer converts a string to a sequence of tokens. Unlike the other languages in this book, μASDL uses a C-like definition of identifiers. It also uses the C++ comment convention: a comment starts with two slashes and goes to the end of the line. If a character is not one of the four reserved characters and is not the start of a constructor or a name, then either it is the start of a comment or it is invalid.

S273a. ⟨*lexical analysis for μASDL* S272b⟩+≡ (S272) ◁S272b

```
asdlToken : token lexer
```

```
val asdlToken =
  let fun commentOrInvalid NONE = NONE
        | commentOrInvalid (SOME (c, cs)) =
            case (c, streamGet cs)
              of (#"/", SOME (#"/", _)) => NONE (* comment to end of line *)
               | _ =>
                   let val msg = "invalid initial character in `" ^
                                 implode (c::listOfStream cs) ^ "'"
                   in  SOME (ERROR msg, EOS)
                   end

      fun or_ p c = c = #"_" orelse p c
      val alpha    = sat (or_ Char.isAlpha)    one
      val alphanum = sat (or_ Char.isAlphaNum) one
      fun constrOrName cs =
        (if List.all (or_ Char.isUpper) cs then CONSTR else NAME)
        (implode cs)
      val token =
            RESERVED <$> sat (Char.contains ",*=|") one
        <|> constrOrName <$> (curry op :: <$> alpha <*> many alphanum)
        <|> (commentOrInvalid o streamGet)
  in  whitespace *> bracketLexer token
  end
```

J.2 A LIST UTILITY

The C code that μASDL generates is full of separators: for example, a function's arguments are separated by commas; assignments are separated by line breaks; and declarations are also separated by line breaks. Separators are inserted using the utility function `foldr1`. Function `foldr1` is a bit like the standard `foldr`, except that it inserts a binary operator *between* elements of a list. When a list contains a single element, `foldr1` returns that element unchanged. When a list is empty, and only then, `foldr1` uses its second argument.

S273b. ⟨*parsers for μASDL* S273b⟩≡ (S272a) S274b ▷

```
foldr1 : ('a * 'a -> 'a) -> 'a -> 'a list -> 'a
```

```
fun foldr1 f z [] = z
  | foldr1 f _ [x] = x
  | foldr1 f z (x::xs) = f (x, foldr1 f z xs)
```

J.3 Abstract syntax and parsing

A μASDL file contains a sequence of definitions. The left-hand side of a definition gives the name of the type being defined, and it marks whether the thing being defined is a pointer type. The right-hand side specifies a sum type or a product type.

S274a. ⟨*abstract syntax for μASDL* S274a⟩≡ (S272a)

```
type def
defName : def -> name
```

```
type name = string
type ty   = string

type     lhs = name * {ptr:bool}
datatype rhs = SUM     of alt list
             | PRODUCT of arg list
     and alt = ALT     of name * arg list option
withtype def = lhs * rhs
     and arg = name * ty
fun defName ((n, _), _) = n
```

In the concrete syntax, an argument to a value constructor or a field of a product type is specified by a sequence of tokens like `char *` or `char *name`. Because the name might be omitted, that syntax can't be represented as an `arg`. Instead, I define type `pre_arg`, which is like an `arg` except the name is optional. Function `preArg` turns a sequence of tokens into a `pre_arg`. Because of C types like `struct Exp`, a `ty` can be formed from more than one token. Multiple tokens are separated by spaces.

S274b. ⟨*parsers for μASDL* S273b⟩+≡ (S272a) ◁S273b S274c▷

```
type pre_arg
preArg : string list -> pre_arg error
```

```
type pre_arg = name option * ty
fun preArg [x] = OK (NONE, x)
  | preArg strings =
      case reverse strings
        of tys as "*" :: _         => OK (NONE,      space (reverse tys))
         | name :: tys             => OK (SOME name, space (reverse tys))
         | [] => ERROR "Empty argument"
and space tys = foldr1 (fn (s, s') => s ^ " " ^ s') "" tys
```

If a constructor carries multiple fields or arguments, every one must be named. This requirement is enforced by function `nameRequired`. The function is Curried so that I can partially apply it, then pass the result to `map`.

S274c. ⟨*parsers for μASDL* S273b⟩+≡ (S272a) ◁S274b S274d▷

```
nameRequired : string -> pre_arg -> arg error
```

```
fun nameRequired thing (SOME x, tau) = OK (x, tau)
  | nameRequired thing (NONE, tau) =
      ERROR ("All arguments of " ^ thing ^ " must be named")
```

A constructor carries an optional list of arguments, and for each argument, a name is also optional. If there is only one argument, and if it has no name, the argument gets the same name as the constructor, except forced to all lower case. If there is more than one argument, *all* the arguments have to have names.

S274d. ⟨*parsers for μASDL* S273b⟩+≡ (S272a) ◁S274c S275b▷

```
toAlt : name -> pre_arg list option -> alt error
```

```
fun toAlt c (NONE) = OK (ALT (c, NONE))
  | toAlt c (SOME args) =
      let fun nameArgs [(NONE, tau)] = OK [(lower c, tau)]
            | nameArgs args = errorList (map (nameRequired c) args)
      in  nameArgs args >>=+ (fn args => ALT (c, SOME args))
      end
```

S275a. ⟨*utility functions for string manipulation and printing* S275a⟩≡

```
val lower = String.map Char.toLower
val upper = String.map Char.toUpper
```

Parsers for individual tokens are stylized.

S275b. ⟨*parsers for μASDL* S273b⟩+≡ (S272a) ◁S274d S275c▷

```
name : name parser
```

```
type 'a parser = (token, 'a) polyparser
val token : token parser = token (* make it monomorphic *)
val pretoken    = (fn (PRETOKEN p) => SOME p | _ => NONE) <$>? token
val name        = (fn (NAME     n) => SOME n | _ => NONE) <$>? pretoken
val constructor = (fn (CONSTR   c) => SOME c | _ => NONE) <$>? pretoken
val reservedChar= (fn (RESERVED c) => SOME c | _ => NONE) <$>? pretoken

fun reserved c = eqx c reservedChar
```

Higher-order functions `commas` and `bars` build parsers for inputs separated by commas or bars.

S275c. ⟨*parsers for μASDL* S273b⟩+≡ (S272a) ◁S275b S275d▷

```
commas : 'a parser -> 'a list parser
bars   : 'a parser -> 'a list parser
```

```
fun commas p = curry op :: <$> p <*> many (reserved #"," *> p)
fun bars   p = curry op :: <$> p <*> many (reserved #"|" *> p)
```

μASDL uses the `bracket` function defined in Appendix I, but it accepts only round brackets, as recognized by `leftRound`.

S275d. ⟨*parsers for μASDL* S273b⟩+≡ (S272a) ◁S275c S275e▷

```
leftRound : bracket_shape parser
```

```
fun leftRound tokens =
  let fun check (_, ROUND) = OK ROUND
        | check (loc, shape) =
            synerrorAt ("don't use " ^ leftString shape ^ "; use (") loc
  in  (check <$>! left) tokens
  end
```

The remaining parsers work up to the most important nonterminals: `alt`, `arg`, `rhs`, and `def`.

S275e. ⟨*parsers for μASDL* S273b⟩+≡ (S272a) ◁S275d

```
product : pre_arg list -> rhs error
alt : alt     parser
arg : pre_arg parser
rhs : rhs     parser
def : def     parser
```

```
fun product preArgs =
  errorList (map (nameRequired "defined type") preArgs) >>=+ PRODUCT

val ptr   =  (fn t => { ptr = isSome t}) <$> optional (reserved #"*")
val arg   =  preArg <$>! (many (name <|> "*" <$ reserved #"*"))
val type' =  pair <$> name <*> ptr
val args  =  leftRound <&> bracket ("(arg, ...)", commas (arg <?> "arg"))
val alt   =  toAlt <$> constructor <*>! optional args
val rhs   =  product <$>! args
         <|> SUM <$> bars alt
val def   =  pair <$> type' <*> (reserved #"=" *> rhs)
```

J.4 Interface to a general-purpose prettyprinter

μASDL generates C code with reasonable indentation and line breaks. Indentation and line breaks are chosen by a technology called *prettyprinting*. The problem has a long history (Oppen 1980; Hughes 1995; Wadler 2003). The code here is based on Christian Lindig's adaptation of Wadler's prettyprinter.

The prettyprinter's central abstraction is the *document*, of type `doc`. A document can be made from any string. Subdocuments may be concatenated (`^^`) to form larger documents, and subdocuments may also be indented. (Indentation is relative to surrounding documents.) And a `brk` document marks a spot where a line break may be introduced.

Concatenation and indentation are related by these laws:

S276a. ⟨*algebraic laws for the prettyprinting combinators* S276a⟩≡ S276b ▷

```
type doc
doc    : string -> doc
^^     : doc * doc -> doc
empty  : doc
indent : int * doc -> doc
brk    : doc
```

```
doc (s ^ t) = doc s ^^ doc t
doc ""      = empty
empty ^^ d  = d
d ^^ empty  = d

indent (0, d)             = d
indent (i, indent (j, d)) = indent (i+j, d)
indent (i, doc s)         = doc s
indent (i, d ^^ d')       = indent (i, d) ^^ indent (i, d')
```

Layout is also governed by laws:

S276b. ⟨*algebraic laws for the prettyprinting combinators* S276a⟩+≡ ◁ S276a

```
layout : int -> doc -> string
```

```
layout (d ^^ d')         = layout d ^ layout d'
layout empty             = ""
layout (doc s)           = s
layout (indent (i, brk)) = "\n" ^ copyChar i " "
```

The last law, together with the laws for `indent`, are the keys to understanding the prettyprinter: `indent` affects *only* what happens to `brk`. In other words, strings aren't indented; instead, indentation is attached to line breaks.

And the last law for `layout` is a bit of a lie; the truth about `brk` is that it is not *always* converted to a newline (plus indentation):

- When `brk` is in a *vertical group*, it always converts to a newline followed by the number of spaces specified by its indentation.
- When `brk` is in a *horizontal group*, it never converts to a newline; instead it converts to a space.
- When `brk` is in an *automatic group*, it converts to a space only if the entire group will the width available; otherwise the `brk`, and *all* `brk`s in the group, convert to newline-indents.
- When `brk` is in a *fill group*, it *might* convert to a space. Each `brk` is free to convert to newline-indent or to space independently of all the other `brk`s; the layout engine uses only as many newlines as are needed to fit the text into the space available.

Groups are created by grouping functions, and for convenience I add a line-breaking concatenate (`^/`) and some support for adding breaks and semicolons:

S277a. ⟨*prettyprinting combinators* S277a⟩≡ (S272a)

```
⟨definition of doc and functions S285a⟩
infix 2 ^/
fun l ^/ r = l ^^ brk ^^ r
fun addBrk d = d ^^ brk
val semi = doc ";"
fun addSemi d = d ^^ semi
```

```
vgrp    : doc -> doc
hgrp    : doc -> doc
agrp    : doc -> doc
fgrp    : doc -> doc
^/      : doc * doc -> doc
addBrk  : doc -> doc
semi    : doc
addSemi : doc -> doc
```

J.5 C TYPES

The main C types that μASDL needs to generate definitions for are

- Structs and unions, which represent products and sums
- Enumerations, which tag alternatives in a sum
- Pointer types
- Opaque named types (`CTY`)
- “Named” types, which behave just like unnamed types, except μASDL emits typedefs for them.

A “field” of a struct or union has a type and a name. It also does double duty as an argument to a function.

S277b. ⟨*C types* S277b⟩≡ (S272a) S277c ▷

```
fieldName : field -> name
```

```
type kind  = string (* struct or union *)
type tag   = string (* struct, union, or enum tag *)
datatype ctype
  = SU     of kind * tag option * field list  (* struct or union *)
  | ENUM   of tag option * name list
  | PTR    of ctype
  | CTY    of string
  | NAMED  of typedef
and        field = FIELD of ctype * name
withtype typedef = ctype * name
fun fieldName (FIELD (_, f)) = f
```

Named types can be extracted so μASDL can emit `typedefs`:

S277c. ⟨*C types* S277b⟩+≡ (S272a) ◁ S277b S278a ▷

```
namedTypes : ctype -> typedef list
```

```
fun namedTypes tau =
  let fun walk (NAMED (ty, name))  tail = walk ty ((ty, name)::tail)
        | walk (SU (_, _, fields)) tail = foldr addField tail fields
        | walk (PTR ty)            tail = walk ty tail
        | walk (CTY  _)            tail = tail
        | walk (ENUM _)            tail = tail
      and addField (FIELD (ty, _), tail) = walk ty tail
  in  walk tau []
  end
```

Tagged types, which must be defined exactly once, can also be extracted.

S278a. ⟨*C types* S277b⟩+≡ (S272a) ◁S277c

```
taggedTypes : ctype -> ctype list
```

```
fun taggedTypes tau =
  let fun walk (NAMED (ty, _))              tail = walk ty tail
        | walk (t as SU (_, SOME _, fields)) tail = foldr addField (t::tail) fields
        | walk (t as SU (_, NONE,   fields)) tail = foldr addField tail      fields
        | walk (PTR ty)                      tail = walk ty tail
        | walk (CTY  _)                      tail = tail
        | walk (t as ENUM (SOME _, _))       tail = t :: tail
        | walk (ENUM (NONE, _))              tail = tail
      and addField (FIELD (ty, _), tail) = walk ty tail
  in  walk tau []
  end
```

J.6 Prettyprinting C types

I define two ways of prettyprinting a C type:

- The *short* method refers to a `struct`, `union`, or `enum` by its tag, omitting the fields.
- The *long* method includes the fields of a `struct`, `union`, or `enum`.

The long method is used for definitions, and the short method is used for everything else. The functions are mutually recursive, so they go into one big nest.

S278b. ⟨*prettyprinting C types* S278b⟩≡ (S272a) S278c ▷

```
shortTypeDoc : ctype -> doc
longTypeDoc  : ctype -> doc
fieldDoc     : field -> doc
```

```
fun shortTypeDoc (SU   (kind, SOME n, _)) = doc (kind ^ " " ^ n)
  | shortTypeDoc (ENUM (SOME n, _))       = doc ("enum" ^ " " ^ n)
  | shortTypeDoc (PTR ty)                 = shortTypeDoc ty ^^ doc " *"
  | shortTypeDoc (CTY ty)                 = doc ty
  | shortTypeDoc (NAMED (_, name))        = doc name
  | shortTypeDoc (t as (SU (_, NONE, _))) = longTypeDoc t
  | shortTypeDoc (t as ENUM (NONE, _))    = longTypeDoc t
```

When μASDL writes a field declaration, I want the code to look nice, so if the type ends in a star (i.e., it's a pointer type), μASDL doesn't put a space between the type and the field name. That way it produces declarations like "`Value v;`" and "`Exp *e;`", but never anything like "`Exp * e;`", which is ugly.

S278c. ⟨*prettyprinting C types* S278b⟩+≡ (S272a) ◁S278b S279a ▷

```
and fieldDoc (FIELD (ty, name)) =
  let fun nonptrSpace (PTR _)  = empty
        | nonptrSpace (CTY ty) =
            (case reverse (explode ty) of #"*" :: _ => empty
                                        | _ => doc " ")
        | nonptrSpace _        = doc " "
  in   shortTypeDoc ty ^^ nonptrSpace ty ^^ doc name
  end
```

A long type declaration includes the literals of `enum`s and the fields of `struct`s and `union`s. Otherwise it's just like a short type declaration. Auxiliary function `embrace` arranges indentation and groups so that a newline after an opening brace has extra indentation, but a newline before a closing brace does not.

S279a. ⟨*prettyprinting C types* S278b⟩+≡ (S272a) ◁S278c S279b▷

```
and longTypeDoc (ENUM (tag, n :: ns)) =
      let val lits = foldl (fn (n, p) => p ^^ doc "," ^/ doc n) (doc n) ns
      in  agrp (doc "enum" ^^ tagDoc tag ^^ doc " " ^^ embrace (fgrp lits))
      end
  | longTypeDoc (SU (kind, tag, fs)) =
      let val fields = foldr1 (op ^/) empty (map (addSemi o fieldDoc) fs)
      in  agrp (doc kind   ^^ tagDoc tag ^^ doc " " ^^ embrace (agrp fields))
      end
  | longTypeDoc (NAMED (ty, _)) = longTypeDoc ty
  | longTypeDoc ty = shortTypeDoc ty

and embrace d = indent(4, doc "{" ^/ d) ^/ doc "}"
and tagDoc (SOME n) = doc (" " ^ n)
  | tagDoc (NONE)   = empty
```

The prototype for a constructor is associated with a constructor name, and it contains a result type, a function name, and a list of arguments. If the list of arguments is empty, the grammar of C requires that the function be given a prototype like `f(void)`. This requirement is met by passing a `void` document to `foldr1`.

S279b. ⟨*prettyprinting C types* S278b⟩+≡ (S272a) ◁S279a

```
type cons_proto
protodoc : cons_proto -> doc
```

```
type cons_proto = name * (ctype * name * field list)

fun protodoc (_, (result, fname, args)) =
  let fun bracket d = doc "(" ^^ d ^^ doc ")"
  in  fieldDoc (FIELD (result, fname)) ^^
      agrp (indent (4, bracket (foldr1 (fn (x, y) => x ^^ doc "," ^/ y)
                                       (doc "void")
                                       (map fieldDoc args))))
  end
```

J.7 Creating C types from sums and products

A defined sum is converted to a tagged union, which means "`struct` containing `enum` and `union`." A defined product is converted to a `struct`.

Because the `ctype` representation is set up to be easy to prettyprint, not to be easy to create, I define convenience functions for creating `struct`, `union`, and pointer types.

S279c. ⟨*converting sums and products to C types* S279c⟩≡ (S272a) S280a▷

```
anonstruct : field list -> ctype
anonunion  : field list -> ctype
struct'    : name * field list -> ctype
union      : name * field list -> ctype
withPtr    : { ptr:bool } * ctype -> ctype
```

```
fun anonstruct fields = SU ("struct", NONE, fields)
fun anonunion  fields = SU ("union",  NONE, fields)
fun struct' (name, fields) = SU ("struct", SOME name, fields)
fun union   (name, fields) = SU ("union",  SOME name, fields)
fun withPtr ({ptr}, ty) = if ptr then PTR ty else ty
```

One function is called `struct'` because `struct` is a reserved word of Standard ML.

An argument can be converted to a field. And if an alternative in a sum carries arguments, a field is reserved to hold those arguments—for a single argument, a single field, and for multiple arguments, a structure containing them all.

S280a. ⟨*converting sums and products to C types* S279c⟩+≡ (S272a) ◁S279c S280b▷

```
argToField       : arg -> field
altToFieldOption : alt -> field option
```

```
fun argToField (f, ty) =
      FIELD (CTY ty, f)

fun altToFieldOption (ALT (name, NONE))       = NONE
  | altToFieldOption (ALT (name, SOME []))    = NONE
  | altToFieldOption (ALT (_,    SOME [arg])) = SOME (argToField arg)
  | altToFieldOption (ALT (name, SOME args))  =
      SOME (FIELD (anonstruct (map argToField args), lower name))
```

A product and a sum with a single alternative are treated almost identically: each becomes a structure with fields for the arguments.

- A product has whatever fields are listed as arguments.
- A sum always has two fields: a named enumeration `alt`, which identifies which element of the sum is represented, and an anonymous union, which holds the arguments (if any) carried by each alternative.

Because the enumeration in a sum is named, it should be `typedef`'d.

S280b. ⟨*converting sums and products to C types* S279c⟩+≡ (S272a) ◁S280a S281a▷

```
toCtype   : def -> ctype
```

⟨*definition of function* `camelCase` S281b⟩

```
val altsuffix = "alt"

fun toCtype ((n, ptr), PRODUCT args) =
      withPtr (ptr, struct'(n, map argToField args))
  | toCtype ((n, ptr), SUM alts) =
      let val enumname = n ^ altsuffix
          val enum = NAMED ( ENUM (NONE, map (fn (ALT (n, _)) => n) alts)
                           , enumname
                           )
          val u    = anonunion (List.mapPartial altToFieldOption alts)
      in  withPtr (ptr, struct' (n, [FIELD (enum, altsuffix),
                                     FIELD (u, "")]))
      end
```

Function `List.mapPartial f` applies `f` to a list of values and returns only the results that are not `NONE`. `List.mapPartial` is part of the initial basis of Standard ML.

J.8 Creating constructor functions and prototypes

Because C provides no convenient way of creating values of `struct` types, it's not enough just to emit definitions of the types: μASDL also emits *constructor functions* for creating values of the types. A `PRODUCT` needs just a single constructor function. A `SUM` needs a constructor function for each alternative in the sum. And each constructor function needs a prototype.

S281a. ⟨*converting sums and products to C types* S279c⟩+≡ (S272a) ◁S280b

```
toConsProtos : def -> cons_proto list
```

```
fun toConsProtos (lhs as (n, {ptr}), rhs) =
  let val struct_ty = CTY ("struct " ^ n)
      val result_ty = if ptr then NAMED (PTR struct_ty, n) else CTY n
      fun toConsProto suffix rty (ALT (altname, args)) =
            (altname, (rty, "mk" ^ camelCase altname ^ suffix,
                       map argToField (getOpt (args, []))))
      fun altProtos   alts   suffix ty = map (toConsProto suffix ty) alts
      fun fieldProtos fields suffix ty =
        [toConsProto suffix ty (ALT (n, SOME fields))]
      fun dualProtos protos =
        protos "" result_ty @ (if ptr then protos "Struct" struct_ty else [])
  in  case rhs
        of SUM alts       => dualProtos (altProtos alts)
         | PRODUCT fields => dualProtos (fieldProtos fields)
  end
```

Each constructor function is given a name that starts with `mk` and is followed by the name of the constructor in "camel case:" the first letter is upper case, as is every letter that follows an underscore. Other letters are lower case, and underscores are dropped. For example, `BOOLV` is built by `mkBoolv`, and `USER_METHOD` would be built by `mkUserMethod`.

S281b. ⟨*definition of function* `camelCase` S281b⟩≡ (S280b)

```
fun camelCase n =
  let fun cap (#"_" :: cs) = cap cs
        | cap (c :: cs) = Char.toUpper c :: lower cs
        | cap [] = []
      and lower (#"_" :: cs) = cap cs
        | lower (c :: cs) = Char.toLower c :: lower cs
        | lower [] = []
  in  (implode o cap o explode) n
  end
```

Code that emits code can be complex. The code-emitting code begins with some auxiliary functions. Function `isPtr` tells if a C type is a pointer type, and `defSum` tells if a definition is a sum.

S281c. ⟨*auxiliary functions for emitting a constructor function* S281c⟩≡ (S283a) S282a ▷

```
isPtr  : ctype -> bool
defSum : def   -> bool
```

```
fun isPtr (NAMED (ty, _)) = isPtr ty
  | isPtr (PTR _)         = true
  | isPtr _               = false
fun defSum (_, SUM _     ) = true
  | defSum (_, PRODUCT _) = false
```

The value returned by a constructor function is called the *answer*. An answer is normally called n, but if the name n conflicts with an argument, the answer function adds more n's until it gets a name that doesn't conflict. Value argfields is in scope and contains the fields that represent the arguments to the constructor function.

S282a. ⟨*auxiliary functions for emitting a constructor function* S281c⟩+≡ (S283a) ◁S281c S282b▷

```
argfields : field list
answer    : string
```

```
val answer =
  let fun isArg x =
        List.exists (fn f => fieldName f = x) argfields
      fun answerName x = if isArg x then answerName ("n" ^ x) else x
  in  answerName "n"
  end
```

I'd like to write code that manipulates the answer, but I don't know what the answer is going to be called. Function ans enables me to refer to the answer as % within a string.

S282b. ⟨*auxiliary functions for emitting a constructor function* S281c⟩+≡ (S283a) ◁S282a S282c▷

```
ans : string -> doc
```

```
val ans =
  doc o String.translate (fn #"%" => answer | c => str c)
```

Function outerfield names a field of the answer, and innerfield names the subfield of the inner, anonymous union that is associated with an argument (for a sum type only).

S282c. ⟨*auxiliary functions for emitting a constructor function* S281c⟩+≡ (S283a) ◁S282b S282d▷

```
outerfield : name  -> string
innerfield : field -> string
```

```
fun outerfield f =
  answer ^ (if isPtr result then "->" else ".") ^ f
val udot = ""  (* anonymous union; was "u." *)
fun innerfield arg =
  let val single = case argfields of [_] => true | _ => false
      fun select s =
        outerfield (if defSum def then
                      if single then udot ^ s else udot ^ lower cname ^ "." ^ s
                    else s)
  in  select (fieldName arg)
  end
```

Finally, fieldAssignments assigns each argument to a field of the answer.

S282d. ⟨*auxiliary functions for emitting a constructor function* S281c⟩+≡ (S283a) ◁S282c

```
fieldAssignments : doc
```

```
val fieldAssignments =
  let fun assignTo arg = concat [innerfield arg, " = ", fieldName arg, ";"]
  in  foldr1 (op ^/) empty (map (doc o assignTo) argfields)
  end
```

With these auxiliary functions in place, the prettyprinting document that represents the definition of a constructor function is built by `consFunDoc`:

S283a. ⟨*functions that build documents to be emitted* S283a⟩≡ (S272a) S283b ▷

```
                                          consFunDoc : def -> cons_proto -> doc
fun consFunDoc def (proto as (cname, (result, fname, argfields))) =
  let ⟨auxiliary functions for emitting a constructor function S281c⟩
  in  vgrp (protodoc proto ^^ doc " " ^^ embrace (
        fieldDoc (FIELD (result, answer)) ^^ semi ^/  (* declare answer *)
        (if isPtr result then
           ans "% = malloc(sizeof(*%));" ^/           (* allocate answer *)
           ans "assert(% != NULL);" ^^ brk
         else
           empty) ^^
        empty ^/
        (if defSum def then  (* if sum, set tag for this constructor *)
           doc (concat [outerfield altsuffix, " = ", upper cname, ";"]) ^^ brk
         else
           empty) ^^
        fieldAssignments ^/  (* initialize all the fields *)
        ans "return %;"))  (* and return the answer *)
  end
```

J.9 Writing the output

μASDL's output includes chunk definitions for `noweb`. The `root` may be something like `"type definitions"`, the `language` is the language into whose implementation the generated code will be incorporated, and the `name` identifies the exact source of the chunk. (In general a language will have many sets of type definitions; the `name` identifies the source of *these* definitions.)

S283b. ⟨*functions that build documents to be emitted* S283a⟩+≡ (S272a) ◁ S283a S283c ▷

```
fun chunkdefn (root, language, name) =
  let fun defn s = concat ["<<", s, " ((", name, "))>>="]
      fun shared "par" = true
        | shared _     = false
  in  if shared name then defn ("shared " ^ root)
      else                defn (root ^ " for \\" ^ language)
  end
```

A C `typedef` uses the same concrete syntax as a field definition, so `typedefDoc` reuses `fieldDoc`.

S283c. ⟨*functions that build documents to be emitted* S283a⟩+≡ (S272a) ◁ S283b S283d ▷

```
                                          typedefDoc : typedef -> doc
fun typedefDoc (ty, name) =
  agrp (doc "typedef " ^^ fieldDoc (FIELD (ty, name)) ^^ semi)
```

μASDL emits a `typedef` for every definition, plus additional `typedef`s for internal, named types.

S283d. ⟨*functions that build documents to be emitted* S283a⟩+≡ (S272a) ◁ S283c S284a ▷

```
fun typedefs d =
  let val ty = toCtype d
      val typedefs = map typedefDoc ((ty, defName d) :: namedTypes ty)
  in  vgrp (foldr1 (op ^/) empty typedefs) ^^ brk
  end
```

μASDL emits definitions for every tagged type, which in practice includes only `struct` types.

S284a. ⟨*functions that build documents to be emitted* S283a⟩+≡ (S272a) ◁S283d S284b▷

```
fun structDefs d =
  let val defs = map (agrp o addBrk o addSemi o longTypeDoc)
                     (taggedTypes (toCtype d))
  in  vgrp (foldr1 (op ^/) empty defs)
  end
```

For a function declaration, every prototype is followed by a semicolon. A function definition is built by `consFunDoc`. Function definitions are separated by blank lines.

S284b. ⟨*functions that build documents to be emitted* S283a⟩+≡ (S272a) ◁S284a

```
fun constructProto d =
  vgrp (foldr1 (op ^/) empty (map (addSemi o protodoc) (toConsProtos d)))

fun constructorFunction d =
  let val funs = map (consFunDoc d) (toConsProtos d)
  in  vgrp (foldr1 (fn (x, y) => x ^/ empty ^/ y) empty funs) ^^ brk
  end
```

Constructor functions are written to a C file, and definitions of four `noweb` chunks are written to a `.xnw` file.

S284c. ⟨*function* `process`*, which reads input and writes output* S284c⟩≡ (S272a)

```
fun process cname webname name lang defstream =
  let val cfile  = TextIO.openOut cname
      val webout = TextIO.openOut webname
      fun printdoc file s =
            TextIO.output(file, layout 75 (vgrp (agrp s^^brk)))
      val (printc, printw) = (printdoc cfile, printdoc webout)
      val defs = listOfStream defstream
      fun chunk (c, mkDoc) =
          ( printw (doc (chunkdefn (c, lang, name)))
          ; app (printw o mkDoc) defs
          )
  in  ( printc (doc "#include \"all.h\"")
      ; app (printc o constructorFunction) defs

      ; chunk ("type definitions",      typedefs)
      ; chunk ("structure definitions", structDefs)
      ; chunk ("type and structure definitions",
                            (fn d => typedefs d ^^ structDefs d ^^ brk))
      ; chunk ("function prototypes",   constructProto)
      ; app TextIO.closeOut [cfile, webout]
      )
  end
```

J.10 Implementation of the prettyprinter

The prettyprinter is derived from one written by Christian Lindig for the C-- project, which in turn is based on Wadler's (2003) prettyprinter. The definition of doc simply gives the alternatives. The BREAK indicates that a break is permissible, but if the break is not taken, the prettyprinter inserts the selected string instead.

S285a. ⟨*definition of* doc *and functions* S285a⟩≡ (S277a) S285b ▷

```
datatype doc
  = ^^     of doc * doc
  | TEXT   of string
  | BREAK  of string
  | INDENT of int * doc
  | GROUP  of break_line or_auto * doc
```

The grouping mechanism is defined two layers. The inner layer, break_line, includes the three basic ways of deciding whether BREAK should be turned into newline-plus-indentation. The outer layer adds AUTO, which is converted to either YES or NO inside the implementation:

S285b. ⟨*definition of* doc *and functions* S285a⟩+≡ (S277a) ◁ S285a S285c ▷

```
and break_line
  = NO      (* hgrp -- every break is a space *)
  | YES     (* vgrp -- every break is a newline *)
  | MAYBE   (* fgrp -- paragraph fill (break is newline only when needed) *)
and 'a or_auto
  = AUTO    (* agrp -- NO if the whole group fits; otherwise YES *)
  | B of 'a
```

These value constructors can be awkward to use, so I define convenience functions.

S285c. ⟨*definition of* doc *and functions* S285a⟩+≡ (S277a) ◁ S285b S286a ▷

```
val doc    = TEXT
val brk    = BREAK " "
val indent = INDENT
val empty  = TEXT ""
infix 2 ^^

fun hgrp d = GROUP (B NO,    d)
fun vgrp d = GROUP (B YES,   d)
fun agrp d = GROUP (  AUTO,  d)
fun fgrp d = GROUP (B MAYBE, d)
```

The layout function converts a document into a string. But it's easier to define a function that solves a more general problem: convert a *list* of documents, each of which is tagged with a *current indentation* and a *break mode*.[1] Making the input a tagged list makes most of the operations easy:

- When a d ‘d' is removed from the head of the list, d and d' are put back separately.

- When a TEXT s is removed from the head of the list, s is added to the result list.

- When an INDENT (i, d) is removed from the head of the list, it is replaced with d, appropriately tagged with the additional indentation.

[1] And for efficiency, I make the result a list of strings, which are concatenated at the very end. This trick is important because repeated concatenation has costs that are quadratic in the size of the result; the cost of a single concatenation at the end is linear.

- When a BREAK is removed from the head of the list, a newline with indentation may or may not be added to the result, depending on the break mode and the space available.

- When a GROUP(AUTO, d) is removed from the head of the list, d is tagged with either YES or NO, depending on space available, and it is put back on the head of the list.

- When any other kind of GROUP(B mode, d) is removed from the head of the list, d is tagged with mode and is put back on the head of the list.

Function format takes a total line width, the number of characters consumed on the current line, and a list of tagged docs. "Putting an item back on the head of the list" is accomplished with internal function reformat.

S286a. ⟨*definition of* doc *and functions* S285a⟩+≡ (S277a) ◁S285c S286b▷

```
format : int -> int -> (int * break_line * doc) list -> string list
```

```
fun format w k [] = []
  | format w k (tagged_doc :: z) =
      let fun reformat item = format w k (item::z)
          fun copyChar 0 c = []
            | copyChar n c = c :: copyChar (n - 1) c
          fun addString s = s :: format w (k + size s) z
          fun breakAndIndent i =
                implode (#"\n" :: copyChar i #" ") :: format w i z
      in  case tagged_doc
            of (i,b, d ^^ d')         => format w k ((i,b,d)::(i,b,d')::z)
             | (i,b,TEXT s)           => addString s
             | (i,b,INDENT(j,d))      => reformat (i+j,b,d)
             | (i,NO, BREAK s)        => addString s
             | (i,YES,BREAK _)        => breakAndIndent i
             | (i,MAYBE, BREAK s)     => if fits (w - k - size s, z)
                                           then addString s
                                           else breakAndIndent i
             | (i,b,GROUP(AUTO, d))   => if fits (w - k, (i,NO,d) :: z)
                                           then reformat (i,NO,d)
                                           else reformat (i,YES,d)
             | (i,b,GROUP(B break,d)) => reformat (i,break,d)
      end
```

Decisions about whether space is available are made by the fits function. It looks ahead at a list of documents and says whether *everything* up to the next possible break will fit in w characters.

S286b. ⟨*definition of* doc *and functions* S285a⟩+≡ (S277a) ◁S286a S287▷

```
fits : int * (int * break_line * doc) list -> bool
```

```
and fits (w, []) = w >= 0
  | fits (w, tagged_doc::z) =
      w >= 0 andalso
      case tagged_doc
       of (i, m,     x ^^ y)      => fits (w, (i,m,x)::(i,m,y)::z)
        | (i, m,     TEXT s)      => fits (w - size s, z)
        | (i, m,     INDENT(j,x)) => fits (w, (i+j,m,x)::z)
        | (i, NO,    BREAK s)     => fits (w - size s, z)
        | (i, YES,   BREAK _)     => true
        | (i, MAYBE, BREAK _)     => true
        | (i, m,     GROUP(_,x))  => fits (w, (i,NO,x)::z)
```

If fits reaches a mandatory or optional BREAK before running out of space, the input fits. The interesting policy decision is for GROUP: for purposes of deciding whether to break a line, all groups are considered without line breaks (mode NO). This policy will break a line in an outer group in order to try to keep documents in an inner group together on a single line.

The layout function lays out a single document by converting it to an instance of the more general problem solved by format: wrap the document in an AUTO group (so that lines are broken optionally); tag it in NO-break mode with no indentation; put it in a singleton list; and format it on a line of width w with no characters consumed.

S287. ⟨*definition of* doc *and functions* S285a⟩+≡ (S277a) ◁S286b

```
  fun layout w doc = concat (format w 0 [(0, NO, GROUP (AUTO, doc))])
```

Part VI. The Supporting Cast

Appendix K contents

Supporting code for Impcore K

The most interesting parts of the Impcore interpreter are presented in Chapter 1 and Appendices F and G. But there are several parts left over—code that is used in Impcore, is not shared in any other interpreter, and is not involved in parsing:

- Interfaces not worth showing in Chapter 1
- Parts of the interpreter that aren't worth showing in Chapter 1: function `readevalprint`, which evaluates extended definitions; function `main`; primitives `print` and `printu`; and the code that runs unit tests
- Implementations of simple abstractions (names, function environments)
- The functions that are added to the extensible printer and used to print Impcore's expressions, definitions, and so on

These parts don't warrant a lot of organization and description. And they are not all equally worth reading:

- The unit-testing part is interesting; this code determines how unit tests are run and what it means to pass one. (A version for μScheme, which is very similar to this one, appears in Section L.7.) But unit tests are in the bridge languages not because they help you learn about programming languages, but because they help you write interesting programs. So the unit-testing code is relegated to this appendix.
- Likewise, if you want to understand how extended definitions are evaluated, the evaluation is not formalized, but the implementation is here.
- The extensible printer and its printing functions may be of minor interest, if for example you want to write your own printing functions. But once you've seen a couple, you've seen them all.
- The `Name` abstraction is a classic, and if you've never seen anything like it, you might enjoy reading the code.
- The implementation of function environments is of no interest—it's exactly like the implementation of `Valenv` in Section 1.6.3, only for functions instead of values.

K.1 Additional interfaces

K.1.1 *Creator functions for abstract syntax*

The code that generates support for abstract-syntax trees (Appendix J) generates two creator functions for each form of tree. The creators for the definition forms are declared as follows:

S292a. ⟨*function prototypes for Impcore* S292a⟩≡ (S295a) S293e ▷

```
Userfun mkUserfun(Namelist formals, Exp body);
Def mkVal(Name name, Exp exp);
Def mkExp(Exp exp);
Def mkDefine(Name name, Userfun userfun);
struct Def mkValStruct(Name name, Exp exp);
struct Def mkExpStruct(Exp exp);
struct Def mkDefineStruct(Name name, Userfun userfun);
```

K.1.2 *Extended definitions*

As discussed in the sidebar on page 24, Impcore has not only *true definitions* but also *extended definitions*, which include unit tests. The extended definitions are there to made coding easier; you needn't worry about how they are implemented. But if you are curious, you'll need to start with these descriptions, which specify how extended definitions (including unit tests) are represented:

S292b. ⟨*xdef.t* S292b⟩≡

```
XDef*     = DEF          (Def)
          | USE          (Name)
          | TEST         (UnitTest)
UnitTest* = CHECK_EXPECT (Exp check, Exp expect)
          | CHECK_ASSERT (Exp)
          | CHECK_ERROR  (Exp)
```

Every unit test in a file is stored in a list of type `UnitTestlist`.

S292c. ⟨*type definitions for Impcore* S292c⟩≡ (S295a) S292d ▷

```
typedef struct UnitTestlist *UnitTestlist; // list of UnitTest
```

A `UnitTestlist` is list of pointers of type `UnitTest`. This naming convention is used in all my C code. List types are manifest, and their definitions are in the lists interface in chunk 45d.

K.1.3 *Lists of expressions*

A type for lists of `Exp`s is also declared here.

S292d. ⟨*type definitions for Impcore* S292c⟩+≡ (S295a) ◁ S292c

```
typedef struct Explist *Explist; // list of Exp
```

K.1.4 *Interface to infrastructure: Streams of definitions*

The details of reading characters and converting them to abstract syntax are interesting, but they are more relevant to study of compiler construction than to study of programming languages. From the programming-language point of view, all we need to know is that the evaluator needs a source of extended definitions, which the parser can provide. The details appear in Appendix F.

A source of extended definitions is called an `XDefstream`.

S293a. ⟨*shared type definitions* S293a⟩≡ (S295a) S293d ▷

```
typedef struct XDefstream *XDefstream;
```

To obtain the next definition from such a source, function `readevalprint` calls `getxdef` (chunk S296a). Function `getxdef` returns either a pointer to the next definition or, if the source is exhausted, `getxdef` returns the `NULL` pointer. And if there is some problem converting input to abstract syntax, `getxdef` may call `synerror` (chunk S183a).

S293b. ⟨*shared function prototypes* S293b⟩≡ (S295a) S293c ▷

```
XDef getxdef(XDefstream xdefs);
```

A stream of definitions is created from a source of lines. That source can be a string compiled into the program, created by function `stringxdefs`, or it can be an external file, created by `filexdefs`. Either way, error messages will need to refer to the source by name, so each stream-creation function expects a name. And `filexdefs` expects a parameter of type `Prompts`, which tells it whether to prompt for input.

S293c. ⟨*shared function prototypes* S293b⟩+≡ (S295a) ◁ S293b S293g ▷

```
XDefstream stringxdefs(const char *stringname, const char *input);
XDefstream filexdefs  (const char *filename, FILE *input, Prompts prompts);
```

Prompts are either absent or standard.

S293d. ⟨*shared type definitions* S293a⟩+≡ (S295a) ◁ S293a S293f ▷

```
typedef enum Prompts { NOT_PROMPTING, PROMPTING } Prompts;
```

Function `readevalprint` consumes a stream of extended definitions. It evaluates each true definition, remembers each unit test, and calls itself recursively on each use. When the stream of extended definitions is exhausted, `readevalprint` runs the unit tests it has remembered.

S293e. ⟨*function prototypes for Impcore* S292a⟩+≡ (S295a) ◁ S292a S296b ▷

```
void readevalprint(XDefstream s, Valenv globals, Funenv functions, Echo echo_level);
```

The `echo_level` parameter controls whether `readevalprint` prints the values and names of top-level expressions and functions. In Chapter 1, a similar parameter is given to `evaldef`.

S293f. ⟨*shared type definitions* S293a⟩+≡ (S295a) ◁ S293d S293h ▷

```
typedef enum Echo { NOT_ECHOING, ECHOING } Echo;
```

K.1.5 Interface to the extensible printer

The implementations of `print` and `fprint` are *extensible*; adding a new conversion specification is as simple as calling `installprinter`:

S293g. ⟨*shared function prototypes* S293b⟩+≡ (S295a) ◁ S293c S294b ▷

```
void installprinter(unsigned char c, Printer *take_and_print);
```

The conversion specifications shown in Chapter 1 are installed when the interpreter launches (chunk ⟨*install conversion specifications for* `print` *and* `fprint` S304c⟩). The details, including the definition of `Printer`, are in Sections F.4 and K.3.2.

type	`Def`	$\mathcal{A}$
type	`Exp`	$\mathcal{A}$
type	`Funenv`	44e
type	`Name`	43a
type	`Namelist`	43a
type	`Printer`	S177a
type	`Userfun`	$\mathcal{A}$
type	`Valenv`	44e
type	`XDef`	$\mathcal{A}$

K.1.6 Source locations and error signaling

An error message is often associated with a particular location in the source code, of type `Sourceloc`. Values of type `Sourceloc` are created by the parsing infrastructure described in Appendix G, which is the place from which `synerror` is called.

S293h. ⟨*shared type definitions* S293a⟩+≡ (S295a) ◁ S293f S294a ▷

```
typedef struct Sourceloc *Sourceloc;
```

The possibility of printing source-code locations complicates the interface to the error module. When an interpreter is reading code interactively, printing source-code locations is silly—if there's a syntax error, it's in what you just typed. But if the interpreter is reading code from a file, you'd like to know the file's name and the number of the line containing the bad syntax. The error module doesn't know where the interpreter is reading code from—only the `main` function in chunk S297a knows that. So the error module has to be told how syntax errors should be formatted: with locations or without.

S294a. ⟨*shared type definitions* S293a⟩+≡ (S295a) ◁ S293h S300e ▷

```
typedef enum ErrorFormat { WITH_LOCATIONS, WITHOUT_LOCATIONS } ErrorFormat;
```

S294b. ⟨*shared function prototypes* S293b⟩+≡ (S295a) ◁ S293g S300c ▷

```
void set_toplevel_error_format(ErrorFormat format);
```

K.1.7 Organizing interfaces into a header file

C provides poor support for separating interfaces from implementations. The best a programmer can do is put each interface in a `.h` file and use the C preprocessor to `#include` those `.h` files where they are needed. Ensuring that the right files are `#include`'d, that they are `#include`'d in the right order, and that no file is `#include`'d more than once are all up to the programmer; the C language and preprocessor don't help. These problems are common, and C programmers have developed conventions to deal with them, but these conventions are better suited to large software projects than to small interpreters. I have therefore chosen simply to put all the interfaces into one header file, `all.h`. When Noweb extracts code from the book, it automatically puts `#include "all.h"` at the beginning of each C file.

File `all.h`, which includes all interfaces used in the interpreter, is split into these parts:

- Imports of header files from the standard C library
- Type definitions
- Structure definitions
- Function prototypes
- Arcana used in lexical analysis and parsing

Putting types, structures, and functions in that order makes it easy for functions or structures declared in one interface to use types defined in another. And because declarations and definitions of types always precede the function prototypes that use those types, I need not worry about getting things in the right order.

To make it possible to reuse the general-purpose interfaces in multiple interpreters, the definitions and prototypes are split into two groups: shared and unshared. A definition is "shared" if it is used in another interpreter.

S295a. ⟨`all.h` *for Impcore* S295a⟩≡

```
#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <inttypes.h>
#include <limits.h>
#include <setjmp.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

⟨type definitions for Impcore S292c⟩
⟨shared type definitions S293a⟩

⟨structure definitions for Impcore S194b⟩
⟨shared structure definitions S166a⟩

⟨definition of __noreturn S295b⟩
⟨function prototypes for Impcore S292a⟩
⟨shared function prototypes S293b⟩

⟨macro definitions used in parsing S195d⟩
⟨declarations of globals used in lexical analysis and parsing S202e⟩
```

A function's prototype can include an annotation that tells `gcc` or `clang` that the function doesn't return. To hold the annotation, I define a macro `__noreturn`. If the code is compiled with `gcc` or `clang`, the macro expands to the GNU C extension `__attribute__((noreturn))`. If the code is compiled with another C compiler, the macro expands to the empty string—other compilers have to live without the information that the function doesn't return.

S295b. ⟨*definition of* `__noreturn` S295b⟩≡ (S295a)

```
#ifdef __GNUC__
#define __noreturn __attribute__((noreturn))
#else
#define __noreturn
#endif
```

K.2 Additional interpreter components

K.2.1 Evaluator for extended definitions

As shown on page S292, the `XDef` type includes both ordinary and extended definitions, and an `XDefstream` provides a stream of `XDefs`, usually from a file or from a user's input.

Responsibility for evaluating definitions is divided between two functions. Function `readevalprint` takes as input a stream of definitions. The extended definitions are handled directly in `readevalprint`:

- Each unit test is remembered and later run.
- A file mentioned in `use` is converted to a stream of extended definitions, then passed recursively to `readevalprint`.

A true definition is passed on to `evaldef`.

S296a. ⟨*eval.c* S296a⟩≡

```
void readevalprint(XDefstream xdefs, Valenv globals, Funenv functions, Echo echo) {
    UnitTestlist pending_unit_tests = NULL;  // run when xdefs is exhausted

    for (XDef d = getxdef(xdefs); d; d = getxdef(xdefs))
        switch (d->alt) {
        case TEST:
            pending_unit_tests = mkUL(d->test, pending_unit_tests);
            break;
        case USE:
            ⟨evaluate d->use, possibly mutating globals and functions S296c⟩
            break;
        case DEF:
            evaldef(d->def, globals, functions, echo);
            break;
        default:
            assert(0);
        }

    process_tests(pending_unit_tests, globals, functions);
}
```

Function `process_tests`, which is defined in Section K.2.4 (page S300), runs the listed `pending_unit_tests` in the order in which they appear in the source code.

S296b. ⟨*function prototypes for Impcore* S292a⟩+≡ (S295a) ◁ S293e S300b ▷

```
void process_tests(UnitTestlist tests, Valenv globals, Funenv functions);
```

On seeing `use`, `readevalprint` opens the file named by `use`, builds a stream of definitions, and recursively calls itself on that stream. The eventual effect will be to call `evaldef` on every definition in the file. When reading definitions via `use`, the interpreter neither prompts nor echoes.

S296c. ⟨*evaluate* `d->use`*, possibly mutating* `globals` *and* `functions` S296c⟩≡ (S296a)

```
{
    const char *filename = nametostr(d->use);
    FILE *fin = fopen(filename, "r");
    if (fin == NULL)
        runerror("cannot open file \"%s\"", filename);
    readevalprint(filexdefs(filename, fin, NOT_PROMPTING),
                  globals, functions, echo);
    fclose(fin);
}
```

As noted in Exercise 35 (Chapter 1), this code can leak open file descriptors.

K.2.2 Implementation of `main`

The `main` function coordinates all the code and forms a working interpreter. Before entering its main loop, the interpreter initializes itself in three phases:

- It initializes `print` and `fprint` (Section K.3.2).
- It creates empty environments for functions and global variables, then populates the `functions` environment with functions from the initial basis.
- Looking at command-line options, it sets `prompts` and `echoes`, which determine whether the interpreter prints (respectively) a prompt and a result for each definition.

At this point the interpreter is ready to process whatever input or inputs are designated in the remainder of the command line, which are pointed to by `firstpath`.

S297a. ⟨*impcore.c* S297a⟩≡

```
int main(int argc, char *argv[]) {
    ⟨install conversion specifications for print and fprint S304c⟩

    Valenv globals   = mkValenv(NULL, NULL);
    Funenv functions = mkFunenv(NULL, NULL);
    ⟨install the initial basis in functions S297b⟩

    Prompts prompts  = PROMPTING;     // default behaviors
    Echo    echoes   = ECHOING;

    char **firstpath; // pointer to first first pathname in argv (or to NULL)
    ⟨process options, leaving firstpath pointing to the name of the first input file S299a⟩
    set_toplevel_error_format(prompts == PROMPTING
                              ? WITHOUT_LOCATIONS
                              : WITH_LOCATIONS);

    for ( ; *firstpath; firstpath++) {
        ⟨evaluate definitions in file designated by *firstpath S298a⟩
    }
    return 0;
}
```

The initial basis includes both primitives and user-defined functions. The primitives are installed first.

S297b. ⟨*install the initial basis in* `functions` S297b⟩≡ (S297a) S297d ▷

```
{
    static const char *prims[] =
      { "+", "-", "*", "/", "<", ">", "=", "println", "print", "printu", 0 };
    for (const char **p = prims; *p; p++) {
        Name x = strtoname(*p);
        bindfun(x, mkPrimitive(x), functions);
    }
}
```

The user-defined functions are stored in a single string. These functions also appear in Figure 1.3 on page 27, from which this code is derived automatically.

S297c. ⟨*predefined Impcore functions, as strings* S297c⟩≡ (S297d)

```
        "(define and (b c) (if b c b))\n"
        "(define or  (b c) (if b b c))\n"
        "(define not (b)   (if b 0 1))\n"
        "(define <= (x y) (not (> x y)))\n"
        "(define >= (x y) (not (< x y)))\n"
        "(define != (x y) (not (= x y)))\n"
        "(define mod (m n) (- m (* n (/ m n))))\n"
        "(define negated (n) (- 0 n))\n"
```

The string is interpreted by `readevalprint`.

S297d. ⟨*install the initial basis in* `functions` S297b⟩+≡ (S297a) ◁ S297b

```
{
    const char *fundefs =
      ⟨predefined Impcore functions, as strings S297c⟩;
    if (setjmp(errorjmp))
        assert(0); // if error in predefined function, die horribly
    readevalprint(stringxdefs("predefined functions", fundefs),
                  globals, functions, NOT_ECHOING);
}
```

`bindfun`	45b
`type Echo`	S293f
`errorjmp`	S181b
`evaldef`	45c
`filexdefs`	S293c
`type Funenv`	44e
`getxdef`	S293b
`mkFunenv`	44f
`mkPrimitive`	44d
`mkUL`	$\mathcal{A}$
`mkValenv`	44f
`type Name`	43a
`nametostr`	43b
`type Prompts`	S293d
`readevalprint`	S293e
`runerror`	47a
`set_toplevel_error_format`	S294b
`stringxdefs`	S293c
`strtoname`	43b
`type UnitTestlist`	S292c
`type Valenv`	44e
`type XDef`	$\mathcal{A}$
`type XDefstream`	S293a

A file is evaluated by setting `fin` to the file designated by `*firstpath`, and by setting `filename` to its name. The interpreter then calls `filexdefs` to turn `fin` into a stream of extended definitions. As a useful pun, when the path name is `"-"`, it designates standard input, *not* the file named `-` in the current directory. (Such a file can be designated by the path `./-`.)

S298a. ⟨*evaluate definitions in file designated by* `*firstpath` S298a⟩≡ (S297a)

```
FILE *fin = !strcmp(*firstpath, "-") ? stdin : fopen(*firstpath, "r");
⟨if fopen failed, roll over and die S298b⟩
const char *filename = fin == stdin ? "standard input" : *firstpath;
XDefstream xdefs = filexdefs(filename, fin, prompts);
⟨evaluate all the extended definitions in xdefs S298c⟩
if (fin != stdin)
    fclose(fin);
```

Error handling is necessary but uninteresting.

S298b. ⟨*if* `fopen` *failed, roll over and die* S298b⟩≡ (S298a)

```
if (fin == NULL) {
    fprintf(stderr, "%s: cannot open file \"%s\"", argv[0], *firstpath);
    exit(1);
}
```

The main loop is in the `readevalprint` function, the call to which is preceded by a C idiom:

S298c. ⟨*evaluate all the extended definitions in* `xdefs` S298c⟩≡ (S298a)

```
while (setjmp(errorjmp))
    /* error recovery, if needed, would appear here */;
readevalprint(xdefs, globals, functions, echoes);
```

This idiom uses `setjmp` to deal with errors. On the first loop test, `setjmp` initializes `errorjmp` and returns zero, so the body of the `while` loop is not executed, and control continues following the loop. If an error occurs later, the error routine calls `longjmp(errorjmp, 1)`, which returns control to the `setjmp` again, this time returning `1`. At this point the body of the `while` is executed. (If any code were needed to recover from the error, it would appear there, but since no code is needed, the body of the `while` contains just a comment.) Then the `while` condition is evaluated again, and the process starts over from the beginning, because `setjmp` resets the jump buffer and returns zero again.

Code to process options has been a thorn in better sides than mine. The only part of this logic worth discussing is what happens after the last option has been seen:

- A nonempty argument list means to treat every argument as a file name, and to evaluate the extended definitions found in the named file.

- An empty argument list means to evaluate standard input.

The empty argument list is implemented by pointing to a statically allocated list containing only the name `"-"`, which designates standard input.

First the options are scanned in a loop, and then `firstpath` is set.

S299a. ⟨*process options, leaving* `firstpath` *pointing to the name of the first input file* S299a⟩≡ (S297a)

```
(void) argc; // not used
extern void dump_fenv_names(Funenv);
{   char **nextarg = argv+1;
    bool dumped = false;
    while (*nextarg && **nextarg == '-' && (*nextarg)[1] != '\0') {
        ⟨handle option *nextarg S299b⟩
        nextarg++;
    }
    static char *default_paths[] = { "-", NULL };
    static char *no_paths[]      = { NULL };
    firstpath = *nextarg ? nextarg : dumped ? no_paths : default_paths;
}
```

The options themselves are handled by a big conditional.

S299b. ⟨*handle option* `*nextarg` S299b⟩≡ (S299a)

```
if (!strcmp(*nextarg, "-q")) {
    prompts = NOT_PROMPTING;
} else if (!strcmp(*nextarg, "-qq")) {
    prompts = NOT_PROMPTING;
    echoes  = NOT_ECHOING;
} else if (!strcmp(*nextarg, "-names")) {
    dump_fenv_names(functions);
    dumped = true;
    if (nextarg[1]) {
        fprintf(stderr, "Dump options must not take any files\n");
        exit(1);
    }
} else {
    fprintf(stderr, "Usage: impcore [-q|-qq] [pathname ...]\n");
    fprintf(stderr, "       impcore -names\n");
    exit(strcmp(*nextarg, "-help") ? 1 : 0);
}
```

`argc`	S297a
`argv`	S297a
`checkargc`	47c
`echoes`	S297a
`errorjmp`	S181b
`filexdefs`	S293c
`firstpath`	S297a
`functions`	S297a
type `Funenv`	44e
`globals`	S297a
`lengthVL`	$\mathcal{A}$
`nthVL`	$\mathcal{A}$
`print`	S176d
`print_utf8`	S188a
`prompts`	S297a
`readevalprint`	S293e
type `Value`	43c
type `XDefstream`	S293a

K.2.3 Implementations of the printing primitives

The implementations of Impcore's primitive functions `println` and `printu` are so similar to `print` that they are not shown in Chapter 1. Instead, they are here:

S299c. ⟨*apply Impcore primitive* `println` *to* `vs` *and return* S299c⟩≡ (52a)

```
{
    checkargc(e, 1, lengthVL(vs));
    Value v = nthVL(vs, 0);
    print("%v\n", v);
    return v;
}
```

S299d. ⟨*apply Impcore primitive* `printu` *to* `vs` *and return* S299d⟩≡ (52a)

```
{
    checkargc(e, 1, lengthVL(vs));
    Value v = nthVL(vs, 0);
    print_utf8(v);
    return v;
}
```

K.2.4 *Code to run unit tests*

Running a list of unit tests is the job of the function `process_tests`:

S300a. ⟨*imptests.c* S300a⟩≡ S300d ▷

```
void process_tests(UnitTestlist tests, Valenv globals, Funenv functions) {
    set_error_mode(TESTING);
    int npassed = number_of_good_tests(tests, globals, functions);
    set_error_mode(NORMAL);
    int ntests  = lengthUL(tests);
    report_test_results(npassed, ntests);
}
```

Function `number_of_good_tests` runs each test, last one first, and counts the number that pass. So it can catch errors during testing, it expects the error mode to be `TESTING`; calling `number_of_good_tests` when the error mode is `NORMAL` is an *unchecked* run-time error.

S300b. ⟨*function prototypes for Impcore* S292a⟩+≡ (S295a) ◁ S296b S300f ▷

```
int number_of_good_tests(UnitTestlist tests, Valenv globals, Funenv functions);
```

The auxiliary function `report_test_results` prints a report of the results. The reporting code is shared among all interpreters written in C; its implementation appears in Section F.6 (page S185).

S300c. ⟨*shared function prototypes* S293b⟩+≡ (S295a) ◁ S294b S304d ▷

```
void report_test_results(int npassed, int ntests);
```

The list of tests coming in contains the last test first, but the first test is the one that must be run first. Function `number_of_good_tests` therefore recursively runs `tests->tl` before calling `test_result` on `tests->hd`. It returns the number of tests passed.

S300d. ⟨*imptests.c* S300a⟩+≡ ◁ S300a S301a ▷

```
int number_of_good_tests(UnitTestlist tests, Valenv globals, Funenv functions) {
    if (tests == NULL)
        return 0;
    else {
        int n = number_of_good_tests(tests->tl, globals, functions);
        switch (test_result(tests->hd, globals, functions)) {
        case TEST_PASSED: return n+1;
        case TEST_FAILED: return n;
        default:          assert(0);
        }
    }
}
```

If the list `tests` were very long, this recursion might blow the C stack. But the list is only as long as the number of tests written in the source code, I don't expect more than dozens of tests, for which default stack space should be adequate. A huge test suite might have to be broken into multiple files.

The heavy lifting is done by function `test_result`, which returns a value of type `TestResult`.

S300e. ⟨*shared type definitions* S293a⟩+≡ (S295a) ◁ S294a

```
typedef enum TestResult { TEST_PASSED, TEST_FAILED } TestResult;
```

S300f. ⟨*function prototypes for Impcore* S292a⟩+≡ (S295a) ◁ S300b

```
TestResult test_result(UnitTest t, Valenv globals, Funenv functions);
```

Function test_result handles every kind of unit test. In Impcore there are three kinds: check-expect, check-assert, and check-error. Typed languages, starting with Typed Impcore in Chapter 6, have more.

S301a. ⟨*imptests.c* S300a⟩+≡ ◁S300d

```
TestResult test_result(UnitTest t, Valenv globals, Funenv functions) {
    switch (t->alt) {
    case CHECK_EXPECT:
        ⟨run check-expect test t, returning TestResult S301b⟩
    case CHECK_ASSERT:
        ⟨run check-assert test t, returning TestResult S302a⟩
    case CHECK_ERROR:
        ⟨run check-error test t, returning TestResult S302b⟩
    default:
        assert(0);
    }
}
```

A check-expect test is run by running both the “check” and the “expect” expressions, each under the protection of an error handler. If an error occurs under either evaluation, the test fails. Otherwise values check and expect are compared. If they differ, the test fails; if not, the test passes. All failures trigger error messages.

S301b. ⟨*run* check-expect *test* t*, returning* TestResult S301b⟩≡ (S301a)

```
{   Valenv empty_env = mkValenv(NULL, NULL);
    if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_expect.check failed with an error S302d⟩
        bufreset(errorbuf);
        return TEST_FAILED;
    }
    Value check = eval(t->check_expect.check, globals, functions, empty_env);

    if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_expect.expect failed with an error S302e⟩
        bufreset(errorbuf);
        return TEST_FAILED;
    }
    Value expect = eval(t->check_expect.expect, globals, functions, empty_env);

    if (check != expect) {
        ⟨report failure because the values are not equal S302c⟩
        return TEST_FAILED;
    } else {
        return TEST_PASSED;
    }
}
```

bufreset	S174c
check	S302b
empty_env	S302b
errorbuf	S182
eval	45c
type Funenv	44e
lengthUL	𝒜
mkValenv	44f
set_error_mode	S181a
testjmp	S181b
type UnitTest	𝒜
type UnitTestlist	S292c
type Valenv	44e
type Value	43c

A `check-assert` test is run by evaluating just one expression, which should evaluate, without error, to a nonzero value.

S302a. ⟨*run* `check-assert` *test* `t`*, returning* `TestResult` S302a⟩≡ (S301a)

```
{   Valenv empty_env = mkValenv(NULL, NULL);
    if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_assert failed with an error S302g⟩
        bufreset(errorbuf);
        return TEST_FAILED;
    }
    Value v = eval(t->check_assert, globals, functions, empty_env);

    if (v == 0) {
        ⟨report failure because the value is zero S302f⟩
        return TEST_FAILED;
    } else {
        return TEST_PASSED;
    }
}
```

A `check-error` error test is also run by evaluating an expression under the protection of an error handler, but this time, if an error occurs, the test passes. If not, the test fails.

S302b. ⟨*run* `check-error` *test* `t`*, returning* `TestResult` S302b⟩≡ (S301a)

```
{   Valenv empty_env = mkValenv(NULL, NULL);
    if (setjmp(testjmp)) {
        bufreset(errorbuf);
        return TEST_PASSED; // error occurred, so the test passed
    }
    Value check = eval(t->check_error, globals, functions, empty_env);
    ⟨report that evaluating t->check_error produced check S303a⟩
    return TEST_FAILED;
}
```

Error-reporting code is voluminous but uninteresting.

S302c. ⟨*report failure because the values are not equal* S302c⟩≡ (S301b)

```
fprint(stderr, "Check-expect failed: expected %e to evaluate to %v",
       t->check_expect.check, expect);
if (t->check_expect.expect->alt != LITERAL)
    fprint(stderr, " (from evaluating %e)", t->check_expect.expect);
fprint(stderr, ", but it's %v.\n", check);
```

S302d. ⟨*report that evaluating* `t->check_expect.check` *failed with an error* S302d⟩≡ (S301b)

```
fprint(stderr, "Check-expect failed: expected %e to evaluate to the same "
               "value as %e, but evaluating %e causes an error: %s.\n",
               t->check_expect.check, t->check_expect.expect,
               t->check_expect.check, bufcopy(errorbuf));
```

S302e. ⟨*report that evaluating* `t->check_expect.expect` *failed with an error* S302e⟩≡ (S301b)

```
fprint(stderr, "Check-expect failed: expected %e to evaluate to the same "
               "value as %e, but evaluating %e causes an error: %s.\n",
               t->check_expect.check, t->check_expect.expect,
               t->check_expect.expect, bufcopy(errorbuf));
```

S302f. ⟨*report failure because the value is zero* S302f⟩≡ (S302a)

```
fprint(stderr, "Check-assert failed: %e evaluated to 0.\n", t->check_assert);
```

S302g. ⟨*report that evaluating* `t->check_assert` *failed with an error* S302g⟩≡ (S302a)

```
fprint(stderr, "Check-assert failed: evaluating %e causes an error: %s.\n",
               t->check_assert, bufcopy(errorbuf));
```

S303a. ⟨*report that evaluating* `t->check_error` *produced* `check` S303a⟩≡ (S302b)

```
fprint(stderr, "Check-error failed: evaluating %e was expected to produce "
               "an error, but instead it produced the value %v.\n",
               t->check_error, check);
```

K.3 Implementations of other abstractions

K.3.1 Implementation of function environments

This code is continued from Chapter 1, which gives the implementation of value environments. Except for the types, the implementation of function environments is identical to code in Section 1.6.3 (page 54).

S303b. ⟨*env.c* S303b⟩≡ S303c ▷

```
struct Funenv {
    Namelist xs;
    Funclist funs;
    // invariant: both lists are the same length
};
```

S303c. ⟨*env.c* S303b⟩+≡ ◁ S303b S303d ▷

```
Funenv mkFunenv(Namelist xs, Funclist funs) {
    Funenv env = malloc(sizeof *env);
    assert(env != NULL);
    assert(lengthNL(xs) == lengthFL(funs));
    env->xs = xs;
    env->funs = funs;
    return env;
}
```

S303d. ⟨*env.c* S303b⟩+≡ ◁ S303c S303e ▷

```
static Func* findfun(Name name, Funenv env) {
    Namelist xs   = env->xs;
    Funclist funs = env->funs;

    for ( ; xs && funs; xs = xs->tl, funs = funs->tl)
        if (name == xs->hd)
            return &funs->hd;
    return NULL;
}
```

S303e. ⟨*env.c* S303b⟩+≡ ◁ S303d S303f ▷

```
bool isfunbound(Name name, Funenv env) {
    return findfun(name, env) != NULL;
}
```

S303f. ⟨*env.c* S303b⟩+≡ ◁ S303e S304a ▷

```
Func fetchfun(Name name, Funenv env) {
    Func *fp = findfun(name, env);
    assert(fp != NULL);
    return *fp;
}
```

S304a. ⟨*env.c* S303b⟩+≡ ◁S303f S304b▷

```
void bindfun(Name name, Func fun, Funenv env) {
    Func *fp = findfun(name, env);
    if (fp != NULL)
        *fp = fun;              // safe optimization
    else {
        env->xs   = mkNL(name, env->xs);
        env->funs = mkFL(fun,  env->funs);
    }
}
```

S304b. ⟨*env.c* S303b⟩+≡ ◁S304a

```
void dump_fenv_names(Funenv env) {
    Namelist xs;
    if (env)
        for (xs = env->xs; xs; xs = xs->tl)
            print("%n\n", xs->hd);
}
```

K.3.2 Initialization of the extensible printer

Table 1.6 (page 47) lists all the types of values that `print`, `fprint`, `runerror`, and `synerror` know how to print. Each of the conversion specifiers mentioned in that table has to be installed. They are installed here:

S304c. ⟨*install conversion specifications for* `print` *and* `fprint` S304c⟩≡ (S297a)

```
installprinter('c', printchar);
installprinter('d', printdecimal);
installprinter('e', printexp);
installprinter('E', printexplist);
installprinter('f', printfun);
installprinter('n', printname);
installprinter('N', printnamelist);
installprinter('p', printpar);
installprinter('P', printparlist);
installprinter('s', printstring);
installprinter('t', printdef);
installprinter('v', printvalue);
installprinter('V', printvaluelist);
installprinter('%', printpercent);
```

Functions `printdecimal`, `printname`, `printstring`, and `printpercent` are defined in Section F.4.3 (page S179). Functions that print lists are generated automatically. The remaining functions, which print Impcore's abstract syntax and values, are defined below.

K.3.3 Printing functions for Impcore syntax and values

The following printing functions are specialized for Impcore:

S304d. ⟨*shared function prototypes* S293b⟩+≡ (S295a) ◁S300c

```
Printer printexp, printdef, printvalue, printfun;
```

Function printexp reverses the process of parsing: it renders abstract syntax into concrete syntax.

S305a. ⟨*printfuns.c* S305a⟩≡ S305b ▷

```
void printexp(Printbuf output, va_list_box *box) {
    Exp e = va_arg(box->ap, Exp);
    if (e == NULL) {
        bprint(output, "<null>");
        return;
    }

    switch (e->alt){
    case LITERAL:
        bprint(output, "%v", e->literal);
        break;
    case VAR:
        bprint(output, "%n", e->var);
        break;
    case SET:
        bprint(output, "(set %n %e)", e->set.name, e->set.exp);
        break;
    case IFX:
        bprint(output, "(if %e %e %e)", e->ifx.cond, e->ifx.truex, e->ifx.falsex);
        break;
    case WHILEX:
        bprint(output, "(while %e %e)", e->whilex.cond, e->whilex.exp);
        break;
    case BEGIN:
        bprint(output, "(begin%s%E)", e->begin?" ":"", e->begin);
        break;
    case APPLY:
        bprint(output, "(%n%s%E)", e->apply.name,
                  e->apply.actuals?" ":"", e->apply.actuals);
        break;
    }
}
```

Function printdef works similarly.

S305b. ⟨*printfuns.c* S305a⟩+≡ ◁ S305a S306a ▷

```
void printdef(Printbuf output, va_list_box *box) {
    Def d = va_arg(box->ap, Def);
    if (d == NULL) {
        bprint(output, "<null>");
        return;
    }

    switch (d->alt) {
    case VAL:
        bprint(output, "(val %n %e)", d->val.name, d->val.exp);
        break;
    case EXP:
        bprint(output, "%e", d->exp);
        break;
    case DEFINE:
        bprint(output, "(define %n (%N) %e)", d->define.name,
                  d->define.userfun.formals,
              d->define.userfun.body);
        break;
    }
}
```

Although it's not bound to any conversion specifier, I also define a function that prints extended definitions.

S306a. ⟨*printfuns.c* S305a⟩+≡ ◁S305b S306c▷

```
void printxdef(Printbuf output, va_list_box *box) {
    XDef d = va_arg(box->ap, XDef);
    if (d == NULL) {
        bprint(output, "<null>");
        return;
    }

    switch (d->alt) {
    case USE:
        bprint(output, "(use %n)", d->use);
        break;
    case TEST:
        ⟨print unit test d->test to file output S306b⟩
        break;
    case DEF:
        bprint(output, "%t", d->def);
        break;
    }
    assert(0);
}
```

S306b. ⟨*print unit test* d->test *to file* output S306b⟩≡ (S306a)

```
{   UnitTest t = d->test;
    switch (t->alt) {
    case CHECK_EXPECT:
        bprint(output, "(check-expect %e %e)",
               t->check_expect.check, t->check_expect.expect);
        break;
    case CHECK_ASSERT:
        bprint(output, "(check-assert %e)", t->check_assert);
        break;
    case CHECK_ERROR:
        bprint(output, "(check-error %e)", t->check_error);
        break;
    default:
        assert(0);
    }
}
```

Impcore's values are so simple that a value can be rendered as concrete syntax for an integer literal.

S306c. ⟨*printfuns.c* S305a⟩+≡ ◁S306a S307▷

```
void printvalue(Printbuf output, va_list_box *box) {
    Value v = va_arg(box->ap, Value);
    bprint(output, "%d", v);
}
```

In Impcore, a function can't be rendered as concrete syntax. But for debugging, it helps to see something, so I put some information in angle brackets.

S307. ⟨*printfuns.c* S305a⟩+≡ ◁S306c

```
void printfun(Printbuf output, va_list_box *box) {
    Func f = va_arg(box->ap, Func);
    switch (f.alt) {
    case PRIMITIVE:
        bprint(output, "<%n>", f.primitive);
        break;
    case USERDEF:
        bprint(output, "<userfun (%N) %e>", f.userdef.formals, f.userdef.body);
        break;
    default:
        assert(0);
    }
}
```

`bprint`	S176d
type `Func`	$\mathcal{A}$
type `Printbuf`	S174a
type `UnitTest`	$\mathcal{A}$
type `va_list_box`	S177b
type `Value`	43c
type `XDef`	$\mathcal{A}$

Appendix L contents

Supporting code for μScheme L

The stars of the μScheme show are presented in Chapter 2; the supporting cast is here. In addition to code for implementing environments, for parsing μScheme, and for running unit tests, all of which resemble analogous parts of the Impcore interpreter, you'll also find code that helps with some exercises, as well as some that lays groundwork for μScheme+ (Chapter 3).

L.1 Excerpts from the interpreter

L.1.1 Additional representations for syntax and values

The extended definitions and unit tests are the same as in Impcore.

S309a. ⟨*ast.t* S309a⟩≡

```
XDef* = DEF    (Def)
      | USE    (Name)
      | TEST   (UnitTest)

UnitTest* = CHECK_EXPECT (Exp check, Exp expect)
          | CHECK_ASSERT (Exp)
          | CHECK_ERROR  (Exp)
```

Both syntax and values may be put in lists.

S309b. ⟨*type definitions for μScheme* S309b⟩≡ (S318a) S312c ▷

```
typedef struct UnitTestlist  *UnitTestlist;  // list of UnitTest
typedef struct Explist  *Explist;            // list of Exp
```

S309c. ⟨*early type definitions for μScheme* S309c⟩≡ (S318a)

```
typedef struct Valuelist *Valuelist;     // list of Value
```

L.1.2 Additional interfaces

Allocation

Before the first call to `allocate`, a client must call `initallocate`. For reasons that are discussed in Chapter 4, `initallocate` is given a pointer to the environment containing the global variables.

S309d. ⟨*function prototypes for μScheme* S309d⟩≡ (S318a) S309e ▷

```
void initallocate(Env *globals);
```

Values

Before executing any code that refers to `truev` or `falsev`, client code must call `initvalue`.

S309e. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁ S309d S310a ▷

```
void initvalue(void);
```

Read-eval-print loop

As in the Impcore interpreter, a sequence of extended definitions is handled by `readevalprint`. In principle, `readevalprint` ought to look a lot like `evaldef`. In particular, `readevalprint` ought to take an environment and return an environment. But when an error occurs, `readevalprint` doesn't actually return; it calls `synerror` or `runerror`. And if an error occurs, the interpreter shouldn't forget the definitions that preceded it. So instead of returning a new environment, `readevalprint` writes the new environment through an environment *pointer* `envp`, which is passed as a parameter.

S310a. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S309e S310c▷

```
void readevalprint(XDefstream xdefs, Env *envp, Echo echo);
```

Evaluation

The evaluator for a list of expressions has to be declared.

S310b. ⟨*eval.c declarations* S310b⟩≡

```
static Valuelist evallist(Explist es, Env env);
```

Primitives

μScheme's primitives are installed by function `addprimitives`, which mutates an existing environment pointer by adding a binding to each primitive operation.

S310c. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S310a S310d▷

```
void addprimitives(Env *envp);
```

Printing

Functions that print μScheme's syntax and values are declared as follows:

S310d. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S310c S311d▷

```
void printenv    (Printbuf, va_list_box*);
void printvalue  (Printbuf, va_list_box*);
void printexp    (Printbuf, va_list_box*);
void printdef    (Printbuf, va_list_box*);
void printlambda (Printbuf, va_list_box*);
```

L.1.3 The read-eval-print loop

Function `readevalprint` evaluates definitions, updates the environment `*envp`, and remembers unit tests. After all definitions have been read, it runs the unit tests it has remembered. The last test added to `unit_tests` is the one at the front of the list, but the tests must be run in the order in which they appeared in the source code, which is back to front.

S310e. ⟨*evaldef.c* S310e⟩≡

```
void readevalprint(XDefstream xdefs, Env *envp, Echo echo) {
    UnitTestlist pending_unit_tests = NULL;

    for (XDef xd = getxdef(xdefs); xd; xd = getxdef(xdefs)) {
        ⟨lower extended definition xd as needed S311a⟩
        ⟨evaluate extended definition xd in environment *envp S311b⟩
    }

    process_tests(pending_unit_tests, *envp);
}
```

Function `readevalprint` is shared with the interpreter for μScheme+ (Chapter 3). Chapter 3 describes a *lowering* operation, which is used to implement some of the syntax in μScheme+. For plain μScheme, no lowering is needed.

S311a. ⟨*lower extended definition* `xd` *as needed* S311a⟩≡ (S310e)

```
/* not in uScheme */
```

A true definition or a `use` may modify `*envp`. A unit test is merely added to the list of pending tests.

S311b. ⟨*evaluate extended definition* `xd` *in environment* `*envp` S311b⟩≡ (S310e)

```
switch (xd->alt) {
case DEF:
    *envp = evaldef(xd->def, *envp, echo);
    break;
case USE:
    ⟨read in a file and update *envp S311c⟩
    break;
case TEST:
    pending_unit_tests = mkUL(xd->test, pending_unit_tests);
    break;
default:
    assert(0);
}
```

The `DEF` case assigns to `*envp`. As alluded to in the description of the interface to `readevalprint` (on page S310), the assignment ensures that after a successful call to `evaldef`, the new environment is remembered. Even if a later call to `evaldef` exits the loop by calling `runerror`, the assignment to `*envp` won't be forgotten.

The `DEF` code is trickier than the `DEF` code in Impcore: Impcore's `readevalprint` simply mutates the global environment. In μScheme, environments are not mutable, so `readevalprint` mutates `*envp` instead.

A file is read as in Impcore, except that again the environment cannot be mutated, so `use` mutates `*envp` instead. When `readevalprint` calls itself recursively to read a file, it passes the same `envp` it was given.

S311c. ⟨*read in a file and update* `*envp` S311c⟩≡ (S311b)

```
{
    const char *filename = nametostr(xd->use);
    FILE *fin = fopen(filename, "r");
    if (fin == NULL)
        runerror("cannot open file \"%s\"", filename);
    readevalprint(filexdefs(filename, fin, NOT_PROMPTING), envp, echo);
    fclose(fin);
}
```

Unit tests are run by code in Section L.7.

S311d. ⟨*function prototypes for* μ*Scheme* S309d⟩+≡ (S318a) ◁S310d S313c▷

```
void process_tests(UnitTestlist tests, Env rho);
```

`type Echo`	S293f
`echo`	159e
`type Env`	153a
`evaldef`	155a
`type Explist`	S309b
`filexdefs`	S293c
`getxdef`	S293b
`mkUL`	$\mathcal{A}$
`nametostr`	43b
`print`	S176d
`type Printbuf`	S174a
`runerror`	47a
`type UnitTestlist`	S309b
`type va_list_box`	S177b
`type Valuelist`	S309c
`type XDef`	$\mathcal{A}$
`type XDefstream`	S293a

L.1.4 Boring evaluator code

In Chapter 2, `evaldef` uses conditional code to decide whether and what to print. Because that code is too boring to appear in Chapter 2, it appears here.

S311e. ⟨*if* `echo` *calls for printing, print either* `v` *or the bound name* S311e⟩≡ (160a)

```
if (echo == ECHOING) {
    if (d->val.exp->alt == LAMBDAX)
        print("%n\n", d->val.name);
    else
        print("%v\n", v);
}
```

S312a. ⟨*if* echo *calls for printing, print* v S312a⟩≡ (160b)

```
if (echo == ECHOING)
    print("%v\n", v);
```

L.1.5 Primitives

Compared with Impcore, *μ*Scheme has a ton of primitives. They are grouped into these three functions:

S312b. ⟨*shared function prototypes* S312b⟩≡ (S318a) S322c ▷

```
Primitive arith, binary, unary;
```

To define the primitives and to associate each one with its tag and function, I resort to an old C programmer's technique called "X macros." Each primitive appears in file `prim.h` as a macro `xx(`*name*`,` *tag*`,` *function*`)`. I use the same macros with two different definitions of `xx`: one to create an enumeration with distinct tags, and one to install the primitives in an empty environment. There are other initialization techniques that don't require macros, but using X macros ensures there is a single point of truth about the primitives, which helps guarantee that the enumeration type is consistent with the initialization code. (That point of truth is the file `prim.h`.)

S312c. ⟨*type definitions for μScheme* S309b⟩+≡ (S318a) ◁ S309b

```
enum {
  #define xx(NAME, TAG, FUNCTION) TAG,
  #include "prim.h"
  #undef xx
  UNUSED_TAG
};
```

In `addprimitives`, the `xx` macro extends the initial environment.

S312d. ⟨*install primitive functions into* env S312d⟩≡ (S315b)

```
#define xx(NAME, TAG, FUNCTION) \
    env = bindalloc(strtoname(NAME), mkPrimitive(TAG, FUNCTION), env);
#include "prim.h"
#undef xx
```

Arithmetic primitives

The primitives that implement arithmetic are specified as follows:

S312e. ⟨*prim.h* S312e⟩≡ S313b ▷

```
xx("+", PLUS,  arith)
xx("-", MINUS, arith)
xx("*", TIMES, arith)
xx("/", DIV,   arith)
xx("<", LT,    arith)
xx(">", GT,    arith)
```

Except for division, each of μScheme's arithmetic primitives can be implemented by the corresponding C operator. Division requires a more involved implementation; μScheme's division must always round toward minus infinity, but C's division guarantees a rounding direction only when dividing positive operands.

S313a. ⟨*prim.c* S313a⟩≡ S313d ▷

```
static int32_t divide(int32_t n, int32_t m) {
    if (n >= 0)
        if (m >= 0)
            return n / m;
        else
            return -(( n - m - 1) / -m);
    else
        if (m >= 0)
            return -((-n + m - 1) /  m);
        else
            return -n / -m;
}
```

Other binary primitives

The non-arithmetic binary primitives are `cons` and `=`.

S313b. ⟨*prim.h* S312e⟩+≡ ◁ S312e S314b ▷

```
xx("cons", CONS, binary)
xx("=",    EQ,   binary)
```

These primitives are implemented by function `binary`, which delegates to functions `cons` and `equalatoms`.

S313c. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁ S311d S325c ▷

```
Value cons(Value v, Value w);
Value equalatoms(Value v, Value w);
```

S313d. ⟨*prim.c* S313a⟩+≡ ◁ S313a S314a ▷

```
Value binary(Exp e, int tag, Valuelist args) {
    checkargc(e, 2, lengthVL(args));
    Value v = nthVL(args, 0);
    Value w = nthVL(args, 1);

    switch (tag) {
    case CONS:
        return cons(v, w);
    case EQ:
        return equalatoms(v, w);
    default:
        assert(0);
    }
}
```

`checkargc`	47c
`echo`	159e
`type Exp`	$\mathcal{A}$
`lengthVL`	$\mathcal{A}$
`nthVL`	$\mathcal{A}$
`type Primitive`	152b
`print`	S176d
`type Value`	$\mathcal{A}$
`type Valuelist`	S309c

The implementation of μScheme's primitive equality is not completely trivial. Two values are = only if they are the same number, the same boolean, the same symbol, or both the empty list. Because all these values are atoms, I have named the C function equalatoms. A different C function, equalpairs, is used in Section L.7 to implement the equality used to implement check-expect.

S314a. ⟨*prim.c* S313a⟩+≡ ◁ S313d

```
Value equalatoms(Value v, Value w) {
    if (v.alt != w.alt)
        return falsev;

    switch (v.alt) {
    case NUM:
        return mkBoolv(v.num   == w.num);
    case BOOLV:
        return mkBoolv(v.boolv == w.boolv);
    case SYM:
        return mkBoolv(v.sym   == w.sym);
    case NIL:
        return truev;
    default:
        return falsev;
    }
}
```

Unary primitives

The unary primitives include the type predicates, list operations car and cdr, printing primitives, and error.

S314b. ⟨*prim.h* S312e⟩+≡ ◁ S313b

```
xx("boolean?",   BOOLEANP,   unary)
xx("null?",      NULLP,      unary)
xx("number?",    NUMBERP,    unary)
xx("pair?",      PAIRP,      unary)
xx("function?",  FUNCTIONP,  unary)
xx("symbol?",    SYMBOLP,    unary)
xx("car",        CAR,        unary)
xx("cdr",        CDR,        unary)
xx("println",    PRINTLN,    unary)
xx("print",      PRINT,      unary)
xx("printu",     PRINTU,     unary)
xx("error",      ERROR,      unary)
```

Some of these primitives are implemented in Chapter 2 (chunk 162a). The remaining primitives are implemented by these cases:

S314c. ⟨*other cases for unary primitives* S314c⟩≡ (162a) S315a ▷

```
case BOOLEANP:
    return mkBoolv(v.alt == BOOLV);
case NUMBERP:
    return mkBoolv(v.alt == NUM);
case SYMBOLP:
    return mkBoolv(v.alt == SYM);
case PAIRP:
    return mkBoolv(v.alt == PAIR);
case FUNCTIONP:
    return mkBoolv(v.alt == CLOSURE || v.alt == PRIMITIVE);
```

S315a. ⟨*other cases for unary primitives* S314c⟩+≡ (162a) ◁S314c

```
case CDR:
    if (v.alt == NIL)
        runerror("in %e, cdr applied to empty list", e);
    else if (v.alt != PAIR)
        runerror("cdr applied to non-pair %v in %e", v, e);
    return *v.pair.cdr;
case PRINTLN:
    print("%v\n", v);
    return v;
case PRINT:
    print("%v", v);
    return v;
```

L.1.6 Implementation of the interpreter's `main` *procedure*

As in the Impcore interpreter, function `main` processes arguments, initializes the interpreter, and runs the read-eval-print loop.

S315b. ⟨*scheme.c* S315b⟩≡

```
int main(int argc, char *argv[]) {
    ⟨install conversion specifications for print and fprint S315c⟩

    initvalue();    // initalizes truev and falsev
    extendSyntax(); // adds new syntax for extended interpreters

    Env env = NULL;
    initallocate(&env);
    ⟨install primitive functions into env S312d⟩
    Env primenv = env; // capture for later dump
    ⟨install predefined functions into env S316b⟩

    Prompts prompts  = PROMPTING;     // default behaviors
    Echo    echoes   = ECHOING;
    char **firstpath; // pointer to first first pathname in argv (or to NULL)
    ⟨process options, leaving firstpath pointing to the name of the first input file S316e⟩
    set_toplevel_error_format(prompts == PROMPTING
                              ? WITHOUT_LOCATIONS
                              : WITH_LOCATIONS);

    for ( ; *firstpath; firstpath++) {
        ⟨evaluate definitions in file designated by *firstpath S316c⟩
    }
    return 0;
}
```

μScheme's many printers have to be installed.

S315c. ⟨*install conversion specifications for* `print` *and* `fprint` S315c⟩≡ (S315b) S316a ▷

```
installprinter('c', printchar);
installprinter('d', printdecimal);
installprinter('e', printexp);
installprinter('E', printexplist);
installprinter('\\', printlambda);
installprinter('n', printname);
installprinter('N', printnamelist);
installprinter('p', printpar);
installprinter('P', printparlist);
installprinter('r', printenv);
```

S316a. ⟨*install conversion specifications for* `print` *and* `fprint` S315c⟩+≡ (S315b) ◁ S315c

```
installprinter('s', printstring);
installprinter('t', printdef);
installprinter('v', printvalue);
installprinter('V', printvaluelist);
installprinter('%', printpercent);
installprinter('*', printpointer);
```

As in the Impcore interpreter, the C representation of the initial basis is generated automatically from code in ⟨*predefined μScheme functions* S319a⟩.

S316b. ⟨*install predefined functions into* `env` S316b⟩≡ (S315b)

```
const char *fundefs =
    ⟨predefined μScheme functions, as strings (from ⟨predefined μScheme functions 96a⟩)⟩;
if (setjmp(errorjmp))
    assert(0);  // fail if error occurs in predefined functions
readevalprint(stringxdefs("predefined functions", fundefs), &env, NOT_ECHOING);
```

Also as in Impcore, a file is evaluated by setting `fin` to the file designated by `*firstpath`, setting `filename` to its name, calling `filexdefs`, and finally calling `readevalprint` under the protection of `setjmp`.

S316c. ⟨*evaluate definitions in file designated by* `*firstpath` S316c⟩≡ (S315b)

```
FILE *fin = !strcmp(*firstpath, "-") ? stdin : fopen(*firstpath, "r");
⟨if fopen failed, roll over and die S316d⟩
const char *filename = fin == stdin ? "standard input" : *firstpath;
XDefstream xdefs = filexdefs(filename, fin, prompts);
while (setjmp(errorjmp))
    /* error recovery, if needed, would appear here */;
readevalprint(xdefs, &env, echoes);
if (fin != stdin)
    fclose(fin);
```

Error handling is necessary but uninteresting.

S316d. ⟨*if* `fopen` *failed, roll over and die* S316d⟩≡ (S316c)

```
if (fin == NULL) {
    fprintf(stderr, "%s: cannot open file \"%s\"", argv[0], *firstpath);
    exit(1);
}
```

Options are processed as in Impcore.

S316e. ⟨*process options, leaving* `firstpath` *pointing to the name of the first input file* S316e⟩≡ (S315b)

```
(void) argc; // not used
{   char **nextarg = argv+1;
    bool dumped = false;
    while (*nextarg && **nextarg == '-' && (*nextarg)[1] != '\0') {
        ⟨handle option *nextarg S317a⟩
        nextarg++;
    }
    static char *default_paths[] = { "-", NULL };
    static char *no_paths[]      = { NULL };
    firstpath = *nextarg ? nextarg : dumped ? no_paths : default_paths;
}
```

In addition to the options that the Impcore interpreter understands, a μScheme interpreter can also take a `-primitives` option, which dumps just the names of the primitive functions.

S317a. ⟨*handle option* `*nextarg` S317a⟩≡ (S316e)

```
extern void dump_env_names(Env);
if (!strcmp(*nextarg, "-q")) {
    prompts = NOT_PROMPTING;
} else if (!strcmp(*nextarg, "-qq")) {
    prompts = NOT_PROMPTING;
    echoes  = NOT_ECHOING;
} else if (!strcmp(*nextarg, "-names")) {
    dump_env_names(env);
    dumped = true;
    if (nextarg[1]) {
        fprintf(stderr, "Dump options must not take any files\n");
        exit(1);
    }
} else if (!strcmp(*nextarg, "-primitives")) {
    dump_env_names(primenv);
    dumped = true;
    if (nextarg[1]) {
        fprintf(stderr, "Dump options must not take any files\n");
        exit(1);
    }
} else {
    fprintf(stderr, "Usage: uscheme [-q|-qq] [pathname ...]\n");
    fprintf(stderr, "       uscheme -primitives\n");
    fprintf(stderr, "       uscheme -names\n");
    exit(strcmp(*nextarg, "-help") ? 1 : 0);
}
```

`argc`	S315b
`argv`	S315b
`echoes`	S315b
`type Env`	153a
`env`	S315b
`errorjmp`	S181b
`filexdefs`	S293c
`firstpath`	S315b
`installprinter`	S293g
`primenv`	S315b
`printdef`	S304d
`printpercent`	S179c
`printpointer`	S179c
`printstring`	S179c
`printvalue`	S304d
`printvaluelist`	$\mathcal{A}$
`prompts`	S315b
`readevalprint`	S310a
`stringxdefs`	S293c
`type XDefstream`	S293a

L.1.7 Memory allocation

The μScheme interpreter in Chapter 2 allocates new objects using `malloc`, which requires no special initialization or resetting. A more interesting implementation of `initallocate` is found in Appendix N, which supports the garbage-collecting interpreters described in Chapter 4.

S317b. ⟨*loc.c* S317b⟩≡

```
void initallocate(Env *globals) {
    (void)globals;
}
```

L.1.8 The interpreter's header file

As in Impcore, the whole interpreter is served by a single C header file.

S318a. ⟨`all.h` *for μScheme* S318a⟩≡

```
#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <inttypes.h>
#include <limits.h>
#include <setjmp.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef __GNUC__
#define __noreturn __attribute__((noreturn))
#else
#define __noreturn
#endif

⟨early type definitions for μScheme S309c⟩
⟨type definitions for μScheme S309b⟩
⟨shared type definitions 43a⟩

⟨structure definitions for μScheme S322a⟩
⟨shared structure definitions S196b⟩

⟨function prototypes for μScheme S309d⟩
⟨shared function prototypes S312b⟩

⟨macro definitions used in parsing S195d⟩
⟨declarations of globals used in lexical analysis and parsing S202e⟩
```

L.2 μSCHEME CODE NOT INCLUDED IN CHAPTER 2

Not all the μScheme code used in Chapter 2 is defined there. The leftovers are defined here.

Function `sqrt` returns the largest integer that is not greater than the square root of its argument. It implements a pathetic definition of square root, but it works on perfect squares, and it's also useful for testing primality.

S318b. ⟨*definition of* `sqrt` S318b⟩≡ (118a)

```
-> (define sqrt (n)
     (letrec ([find (lambda (r)
                      (if (> (* r r) n) (- r 1) (find (+ r 1))))])
       (find 0)))
```

Unicode code points

μScheme has no string literals; it has only quoted symbols. To print a character that can't appear in a quoted symbol, use one of these code points:

S319a. ⟨*predefined μScheme functions* S319a⟩≡ S319b ▷

```
(val newline      10)   (val left-round    40)
(val space        32)   (val right-round   41)
(val semicolon    59)   (val left-curly   123)
(val quotemark    39)   (val right-curly  125)
                        (val left-square   91)
                        (val right-square  93)
```

Integer functions

The non-primitive integer operations are defined exactly as they would be in Impcore. First, the comparisons.

S319b. ⟨*predefined μScheme functions* S319a⟩+≡ ◁ S319a S319c ▷

```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

Next, `min` and `max`.

S319c. ⟨*predefined μScheme functions* S319a⟩+≡ ◁ S319b S319d ▷

```
(define max (x y) (if (> x y) x y))
(define min (x y) (if (< x y) x y))
```

Finally, negation, modulus, greatest common divisor, and least common multiple.

S319d. ⟨*predefined μScheme functions* S319a⟩+≡ ◁ S319c S320b ▷

```
(define negated (n) (- 0 n))
(define mod (m n) (- m (* n (/ m n))))
(define gcd (m n) (if (= n 0) m (gcd n (mod m n))))
(define lcm (m n) (if (= m 0) 0 (* m (/ n (gcd m n)))))
```

List operations

Nobody should use these operations. I'm not sure why I have kept them. Tradition?

S319e. ⟨*more predefined combinations of* `car` *and* `cdr` S319e⟩≡ S319f ▷

```
(define cddr  (sx) (cdr (cdr  sx)))
(define caaar (sx) (car (caar sx)))
(define caadr (sx) (car (cadr sx)))
(define cadar (sx) (car (cdar sx)))
(define caddr (sx) (car (cddr sx)))
(define cdaar (sx) (cdr (caar sx)))
(define cdadr (sx) (cdr (cadr sx)))
(define cddar (sx) (cdr (cdar sx)))
(define cdddr (sx) (cdr (cddr sx)))
```

S319f. ⟨*more predefined combinations of* `car` *and* `cdr` S319e⟩+≡ ◁ S319e S320a ▷

```
(define caaaar (sx) (car (caaar sx)))
(define caaadr (sx) (car (caadr sx)))
(define caadar (sx) (car (cadar sx)))
(define caaddr (sx) (car (caddr sx)))
(define cadaar (sx) (car (cdaar sx)))
(define cadadr (sx) (car (cdadr sx)))
(define caddar (sx) (car (cddar sx)))
(define cadddr (sx) (car (cdddr sx)))
```

S320a. ⟨*more predefined combinations of* `car` *and* `cdr` S319e⟩+≡ ◁S319f

```
(define cdaaar (sx) (cdr (caaar sx)))
(define cdaadr (sx) (cdr (caadr sx)))
(define cdadar (sx) (cdr (cadar sx)))
(define cdaddr (sx) (cdr (caddr sx)))
(define cddaar (sx) (cdr (cdaar sx)))
(define cddadr (sx) (cdr (cdadr sx)))
(define cdddar (sx) (cdr (cddar sx)))
(define cddddr (sx) (cdr (cdddr sx)))
```

The functions below, by contrast, are silver. ("Gold" would be a variadic `list` function such as would be enabled by completing Exercise 2 in Chapter 5. Or a variadic `list` primitive.)

S320b. ⟨*predefined μScheme functions* S319a⟩+≡ ◁S319d

```
(define list4 (x y z a)         (cons x (list3 y z a)))
(define list5 (x y z a b)       (cons x (list4 y z a b)))
(define list6 (x y z a b c)     (cons x (list5 y z a b c)))
(define list7 (x y z a b c d)   (cons x (list6 y z a b c d)))
(define list8 (x y z a b c d e) (cons x (list7 y z a b c d e)))
```

L.3 Support for testing the examples from Chapter 2

All the examples in the chapters are tested. And in Chapter 2, the examples include not just ⟨*transcript* S320c⟩ but also ⟨*polymorphic-set transcript* 133b⟩. To test the examples that manipulate polymorphic sets, I use a scurvy Noweb trick; I extend the definition of ⟨*transcript* S320c⟩ to include ⟨*polymorphic-set transcript* 133b⟩. By doing the extension here, in an appendix, I expose ⟨*polymorphic-set transcript* 133b⟩ to my testing software while preventing the definitions in ⟨*polymorphic-set transcript* 133b⟩ from interfering with non-polymorphic examples in Chapter 2.

S320c. ⟨*transcript* S320c⟩≡

⟨*polymorphic-set transcript* 133b⟩

L.4 Implementation of μScheme environments

μScheme's environments are significantly different from Impcore environments, but not so dramatically different that it's worth showing their implementation in Chapter 2. The big difference in a μScheme environment is that evaluating a `lambda` expression copies an environment, and that copy can be extended. The possibility of copying rules out the mutate-in-place optimization I use in Impcore's environments, and it militates toward a different representation.

First, and most important, environments are immutable, as you can tell by analyzing the interface in Chapter 2 (page 153). The operational semantics never mutates an environment, and there is really no need, because only locations are mutated. Moreover, if environments could be mutated then it wouldn't be safe to copy them just by copying pointers; using mutable environments would make the evaluation of `lambda` expressions very expensive.

I choose a representation of environments that makes it easy to share and extend them: an environment contains a single binding and a pointer to the rest of the bindings in the environment.

S320d. ⟨*env.c* S320d⟩≡ S321a ▷

```
struct Env {
    Name name;
    Value *loc;
    Env tl;
};
```

A name is looked up by following `tl` pointers.

S321a. ⟨*env.c* S320d⟩+≡ ◁S320d S321b▷

```
Value* find(Name name, Env env) {
    for (; env; env = env->tl)
        if (env->name == name)
            return env->loc;
    return NULL;
}
```

Function `bindalloc` *always* creates a new environment with a new binding. The existing environment is not mutated.

S321b. ⟨*env.c* S320d⟩+≡ ◁S321a S321c▷

```
Env bindalloc(Name name, Value val, Env env) {
    Env newenv = malloc(sizeof(*newenv));
    assert(newenv != NULL);

    newenv->name = name;
    newenv->loc  = allocate(val);
    newenv->tl   = env;
    return newenv;
}
```

Function `bindalloclist` binds names to values in sequence.

S321c. ⟨*env.c* S320d⟩+≡ ◁S321b S321d▷

```
Env bindalloclist(Namelist xs, Valuelist vs, Env env) {
    for (; xs && vs; xs = xs->tl, vs = vs->tl)
        env = bindalloc(xs->hd, vs->hd, env);
    assert(xs == NULL && vs == NULL);
    return env;
}
```

An environment can be printed, which can be useful if you need to debug your code.

S321d. ⟨*env.c* S320d⟩+≡ ◁S321c S321e▷

```
void printenv(Printbuf output, va_list_box *box) {
    char *prefix = " ";

    bprint(output, "{");
    for (Env env = va_arg(box->ap, Env); env; env = env->tl) {
        bprint(output, "%s%n -> %v", prefix, env->name, *env->loc);
        prefix = ", ";
    }
    bprint(output, " }");
}
```

And an environment's names can be printed, which is what the `-names` and `-primitives` options to the interpreter do.

S321e. ⟨*env.c* S320d⟩+≡ ◁S321d

```
void dump_env_names(Env env) {
    for ( ; env; env = env->tl)
        fprint(stdout, "%n\n", env->name);
}
```

L.5 A parser for μScheme

μScheme is parsed using the shift-reduce parsing technology that is also used for Impcore (Appendix G).

L.5.1 Parsing tables and reduce functions

A shift-reduce parser must be able to parse any syntactic component that can appear in a μScheme program. μScheme includes all the components used to parse Impcore, plus a `Value` component that can appear in a quoted S-expression.

S322a. ⟨*structure definitions for μScheme* S322a⟩≡ (S318a)

```
struct Component {
    Exp exp;
    Explist exps;
    Name name;
    Namelist names;
    Value value;
    ⟨fields of μScheme Component added in exercises S324d⟩
};
```

The keyword expressions are specified by a usage table.

S322b. ⟨*parse.c* S322b⟩≡ S323a ▷

```
struct Usage usage_table[] = {
    { ADEF(VAL),           "(val x e)" },
    { ADEF(DEFINE),        "(define fun (formals) body)" },
    { ANXDEF(USE),         "(use filename)" },
    { ATEST(CHECK_EXPECT), "(check-expect exp-to-run exp-expected)" },
    { ATEST(CHECK_ASSERT), "(check-assert exp)" },
    { ATEST(CHECK_ERROR),  "(check-error exp)" },

    { SET,     "(set x e)" },
    { IFX,     "(if cond true false)" },
    { WHILEX,  "(while cond body)" },
    { BEGIN,   "(begin exp ... exp)" },
    { LAMBDAX, "(lambda (formals) body)" },

    { ALET(LET),     "(let ([var exp] ...) body)" },
    { ALET(LETSTAR), "(let* ([var exp] ...) body)" },
    { ALET(LETREC),  "(letrec ([var exp] ...) body)" },
    ⟨μScheme usage_table entries added in exercises S324i⟩
    { -1, NULL }
};
```

Shift functions are as in Impcore, but with two additions: to parse a quoted S-expression, shift function `sSexp` has been added, and to parse bindings in `LETX` forms, `sBindings` has been added.

S322c. ⟨*shared function prototypes* S312b⟩+≡ (S318a) ◁ S312b S324j ▷

```
ParserResult sSexp    (ParserState state);
ParserResult sBindings(ParserState state);
```

Using the new shift functions, the `exptable` used for parsing expressions is defined here:

S323a. ⟨*parse.c* S322b⟩+≡ ◁S322b S323c▷

```
static ShiftFun quoteshifts[] = { sSexp,                stop };
static ShiftFun setshifts[]   = { sName, sExp,          stop };
static ShiftFun ifshifts[]    = { sExp, sExp, sExp,     stop };
static ShiftFun whileshifts[] = { sExp, sExp,           stop };
static ShiftFun beginshifts[] = { sExps,                stop };
static ShiftFun letshifts[]   = { sBindings, sExp,      stop };
static ShiftFun lambdashifts[]= { sNamelist, sExp,      stop };
static ShiftFun applyshifts[] = { sExp, sExps,          stop };
⟨arrays of shift functions added to μScheme in exercises S324e⟩
⟨lowering functions for μScheme+ S340c⟩

struct ParserRow exptable[] = {
  { "set",    ANEXP(SET),     setshifts },
  { "if",     ANEXP(IFX),     ifshifts },
  { "begin",  ANEXP(BEGIN),   beginshifts },
  { "lambda", ANEXP(LAMBDAX), lambdashifts },
  { "quote",  ANEXP(LITERAL), quoteshifts },
  ⟨rows of μScheme's exptable that are sugared in μScheme+ (from chunk 697b)⟩
  ⟨rows added to μScheme's exptable in exercises S324f⟩
  { NULL,     ANEXP(APPLY),   applyshifts }  // must come last
};
```

Four forms are treated specially because in Chapter 2 they are ordinary syntax, but in Chapter 3 (μScheme+) they are syntactic sugar.

S323b. ⟨*rows of μScheme's* `exptable` *that are sugared in μScheme+* **[[uscheme]]** S323b⟩≡

```
  { "while",  ANEXP(WHILEX),  whileshifts },
  { "let",    ALET(LET),      letshifts },
  { "let*",   ALET(LETSTAR),  letshifts },
  { "letrec", ALET(LETREC),   letshifts },
```

In μScheme, a quote mark in the input is expanded to a `quote` expression. The global variable `read_tick_as_quote` so instructs the `getpar` function defined in Appendix F (page S170).

S323c. ⟨*parse.c* S322b⟩+≡ ◁S323a S323d▷

```
bool read_tick_as_quote = true;
```

The codes used in `exptable` tell `reduce_to_exp` how to reduce components to an expression.

S323d. ⟨*parse.c* S322b⟩+≡ ◁S323c S324b▷

```
Exp reduce_to_exp(int code, struct Component *comps) {
  switch(code) {
  case ANEXP(SET):     return mkSet(comps[0].name, comps[1].exp);
  case ANEXP(IFX):     return mkIfx(comps[0].exp, comps[1].exp, comps[2].exp);
  case ANEXP(BEGIN):   return mkBegin(comps[0].exps);
  ⟨cases for reduce_to_exp that are sugared in μScheme+ (from chunk 697b)⟩
  case ANEXP(LAMBDAX): return mkLambdax(mkLambda(comps[0].names,comps[1].exp));
  case ANEXP(APPLY):   return mkApply(comps[0].exp, comps[1].exps);
  case ANEXP(LITERAL): return mkLiteral(comps[0].value);
  ⟨cases for μScheme's reduce_to_exp added in exercises S324g⟩
  }
  assert(0);
}
```

type `Exp`	$\mathcal{A}$
type `Explist`	S309b
`mkApply`	$\mathcal{A}$
`mkBegin`	$\mathcal{A}$
`mkIfx`	$\mathcal{A}$
`mkLambda`	$\mathcal{A}$
`mkLambdax`	$\mathcal{A}$
`mkLiteral`	$\mathcal{A}$
`mkSet`	$\mathcal{A}$
type `Name`	43a
type `Namelist`	43a
type `ParserResult`	S197d
type `ParserState`	S197a
`sExp`	S197f
`sExps`	S197f
type `ShiftFun`	S197e
`sName`	S197f
`sNamelist`	S197f
`stop`	S200a
type `Value`	$\mathcal{A}$

Again, four forms are treated differently in Chapters 2 and 3.

S324a. ⟨*cases for* `reduce_to_exp` *that are sugared in μScheme+* **⟦uscheme⟧** S324a⟩≡

```
case ANEXP(WHILEX):  return mkWhilex(comps[0].exp, comps[1].exp);
case ALET(LET):
case ALET(LETSTAR):
case ALET(LETREC):
    return mkLetx(code+LET-ALET(LET),
                  comps[0].names, comps[0].exps, comps[1].exp);
```

The `xdeftable` is shared with the Impcore parser. Function `reduce_to_xdef` is almost shareable as well, but not quite—the abstract syntax of `DEFINE` is different.

S324b. ⟨*parse.c* S322b⟩+≡ ◁ S323d S325a ▷

```
XDef reduce_to_xdef(int code, struct Component *out) {
    switch(code) {
    case ADEF(VAL):    return mkDef(mkVal(out[0].name, out[1].exp));
    ⟨reduce_to_xdef case for ADEF(DEFINE) (from chunk 697b)⟩
    case ANXDEF(USE):  return mkUse(out[0].name);
    case ATEST(CHECK_EXPECT):
                       return mkTest(mkCheckExpect(out[0].exp, out[1].exp));
    case ATEST(CHECK_ASSERT):
                       return mkTest(mkCheckAssert(out[0].exp));
    case ATEST(CHECK_ERROR):
                       return mkTest(mkCheckError(out[0].exp));
    case ADEF(EXP):    return mkDef(mkExp(out[0].exp));
    ⟨cases for μScheme's reduce_to_xdef added in exercises S324h⟩
    default:           assert(0);  // incorrectly configured parser
    }
}
```

S324c. ⟨`reduce_to_xdef` *case for* `ADEF(DEFINE)` **⟦uscheme⟧** S324c⟩≡

```
case ADEF(DEFINE): return mkDef(mkDefine(out[0].name,
                                      mkLambda(out[1].names, out[2].exp)));
```

The μScheme parser is set up to be extended by adding definitions to the following Noweb chunks.

S324d. ⟨*fields of μScheme* `Component` *added in exercises* S324d⟩≡ (S322a)

```
/* if implementing COND, add a question-answer field here */
```

S324e. ⟨*arrays of shift functions added to μScheme in exercises* S324e⟩≡ (S323a)

```
/* define arrays of shift functions as needed for [[exptable]] rows */
```

S324f. ⟨*rows added to μScheme's* `exptable` *in exercises* S324f⟩≡ (S323a)

```
/* add a row for each new syntactic form of Exp */
```

S324g. ⟨*cases for μScheme's* `reduce_to_exp` *added in exercises* S324g⟩≡ (S323d)

```
/* add a case for each new syntactic form of Exp */
```

S324h. ⟨*cases for μScheme's* `reduce_to_xdef` *added in exercises* S324h⟩≡ (S324b)

```
/* add a case for each new syntactic form of definition */
```

S324i. ⟨*μScheme* `usage_table` *entries added in exercises* S324i⟩≡ (S322b)

```
/* add expected usage for each new syntactic form */
```

Extending syntax likely also requires some initialization, which is done by function `extendSyntax`.

S324j. ⟨*shared function prototypes* S312b⟩+≡ (S318a) ◁ S322c

```
void extendSyntax(void);
```

S324k. ⟨*parse.c* **⟦uscheme⟧** S324k⟩≡

```
void extendSyntax(void) { }
```

L.5.2 New shift functions: S-expressions and bindings

Many shift functions are reused from Impcore's parser (Appendix G). New shift function `sSexp` calls `parsesx` to parse a literal S-expression. The result is stored in a `value` component.

S325a. ⟨*parse.c* S322b⟩+≡ ◁S324b S325b▷

```
ParserResult sSexp(ParserState s) {
    if (s->input == NULL) {
        return INPUT_EXHAUSTED;
    } else {
        Par p = s->input->hd;
        halfshift(s);
        s->components[s->nparsed++].value = parsesx(p, s->context.source);
        return PARSED;
    }
}
```

New shift function `sBindings` calls `parseletbindings` to parse bindings for `LETX` forms. Function `parseletbindings` returns a component that has both `names` and and `exps` fields set.

S325b. ⟨*parse.c* S322b⟩+≡ ◁S325a S325d▷

```
ParserResult sBindings(ParserState s) {
    if (s->input == NULL) {
        return INPUT_EXHAUSTED;
    } else {
        Par p = s->input->hd;
        switch (p->alt) {
        case ATOM:
            usage_error(code_of_name(s->context.name), BAD_INPUT, &s->context);
        case LIST:
            halfshift(s);
            s->components[s->nparsed++] =
                parseletbindings(&s->context, p->list);
            return PARSED;
        }
        assert(0);
    }
}
```

L.5.3 New parsing functions: S-expressions and bindings

Each new shift function is supported by a new parsing function.

S325c. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S313c S333a▷

```
Value parsesx(Par p, Sourceloc source);
struct Component parseletbindings(ParsingContext context, Parlist input);
```

Parsing quoted S-expressions

A quoted S-expression is either an atom or a list.

S325d. ⟨*parse.c* S322b⟩+≡ ◁S325b S326c▷

```
Value parsesx(Par p, Sourceloc source) {
    switch (p->alt) {
    case ATOM: ⟨return p->atom interpreted as an S-expression S326a⟩
    case LIST: ⟨return p->list interpreted as an S-expression S326b⟩
    }
    assert(0);
}
```

Inside a quoted S-expression, an atom is necessarily a number, a Boolean, or a symbol. This parser does not understand dot notation, which in full Scheme is used to write cons cells that are not lists.

S326a. ⟨*return* `p->atom` *interpreted as an S-expression* S326a⟩≡ (S325d)

```
{
    Name n        = p->atom;
    const char *s = nametostr(n);
    char *t;                        // first nondigit in s
    long l = strtol(s, &t, 10);     // value of digits in s, if any
    if (*t == '\0' && *s != '\0')   // s is all digits
        return mkNum(l);
    else if (strcmp(s, "#t") == 0)
        return truev;
    else if (strcmp(s, "#f") == 0)
        return falsev;
    else if (strcmp(s, ".") == 0)
        synerror(source, "this interpreter cannot handle . "
                         "in quoted S-expressions");
    else
        return mkSym(n);
}
```

A quoted list is turned into a μScheme list, recursively.

S326b. ⟨*return* `p->list` *interpreted as an S-expression* S326b⟩≡ (S325d)

```
if (p->list == NULL)
    return mkNil();
else
    return cons(parsesx(p->list->hd, source),
                parsesx(mkList(p->list->tl), source));
```

Parsing bindings used in `LETX` *forms*

A sequence of let bindings has both names and expressions. To capture both, `parseletbindings` returns a component with both `names` and `exps` fields set.

S326c. ⟨*parse.c* S322b⟩+≡ ◁S325d S327a▷

```
struct Component parseletbindings(ParsingContext context, Parlist input) {
    if (input == NULL) {
        struct Component output = { .names = NULL, .exps = NULL };
        return output;
    } else if (input->hd->alt == ATOM) {
        synerror(context->source,
                 "in %p, expected (... (x e) ...) in bindings, but found %p",
                 context->par, input->hd);
    } else {
        /* state and row are set up to parse one binding */
        struct ParserState s = mkParserState(input->hd, context->source);
        s.context = *context;
        static ShiftFun bindingshifts[] = { sName, sExp, stop };
        struct ParserRow row = { .code   = code_of_name(context->name)
                               , .shifts = bindingshifts
                               };
        rowparse(&row, &s);

        /* now parse the remaining bindings, then add the first at the front */
        struct Component output = parseletbindings(context, input->tl);
        output.names = mkNL(s.components[0].name, output.names);
        output.exps  = mkEL(s.components[1].exp,  output.exps);
        return output;
    }
}
```

L.5.4 Parsing atomic expressions

When an atom appears as an expression, it is a Boolean, an integer literal, or a variable.

S327a. ⟨*parse.c* S322b⟩+≡ ◁S326c S336c▷

```
Exp exp_of_atom (Sourceloc loc, Name n) {
    if (n == strtoname("#t"))
        return mkLiteral(truev);
    else if (n == strtoname("#f"))
        return mkLiteral(falsev);

    const char *s = nametostr(n);
    char *t;                      // first nondigit in s, if any
    long l = strtol(s, &t, 10);   // number represented by s, if any
    if (*t != '\0' || *s == '\0') // not a nonempty sequence of digits
        return mkVar(n);
    else if (((l == LONG_MAX || l == LONG_MIN) && errno == ERANGE) ||
             l > (long)INT32_MAX || l < (long)INT32_MIN)
    {
        synerror(loc, "arithmetic overflow in integer literal %s", s);
        return mkVar(n); // unreachable
    } else {  // the number is the whole atom, and not too big
        return mkLiteral(mkNum(l));
    }
}
```

L.6 Implementation of μScheme's value interface

The value interface has special support for Booleans and for unspecified values. As usual, the value interface also has support for printing.

L.6.1 Boolean values and Boolean testing

The first part of the value interface supports Booleans.

S327b. ⟨*value.c* S327b⟩≡ S328a▷

```
bool istrue(Value v) {
    return v.alt != BOOLV || v.boolv;
}

Value truev, falsev;

void initvalue(void) {
    truev  = mkBoolv(true);
    falsev = mkBoolv(false);
}
```

L.6.2 Unspecified values

The interface defines a function to return an unspecified value. "Unspecified" means the evaluator can pick any value it likes. For example, it could just always use NIL. Unfortunately, if the evaluator always returns NIL, careless persons will grow to rely on finding NIL, and they shouldn't. (Ask me how I know.) To foil such carelessness, the evaluator chooses an unhelpful value at random.

S328a. ⟨*value.c* S327b⟩+≡ ◁S327b

```
Value unspecified (void) {
    switch ((rand()>>4) & 0x3) {
        case 0:  return truev;
        case 1:  return mkNum(rand());
        case 2:  return mkSym(strtoname("this value is unspecified"));
        case 3:  return mkPrimitive(ERROR, unary);
        default: return mkNil();
    }
}
```

L.6.3 Printing and values

Most values can be printed using just a few lines of C code, but the code needed to print a closure is long and tedious. When printing a closure, you don't want to see the entire environment that was captured in the closure. You want to see only the parts of the environment that the closure actually depends on—the *free variables* of the `lambda` expression. (See Section 5.6, page 315.)

Finding free variables in an expression

Finding free variables is hard work. I start with a bunch of utility functions on names. Function `nameinlist` says whether a particular `Name` is on a `Namelist`.

S328b. ⟨*printfuns.c* S328b⟩≡ S328c ▷

```
static bool nameinlist(Name n, Namelist xs) {
    for (; xs; xs=xs->tl)
        if (n == xs->hd)
            return true;
    return false;
}
```

Function `addname` adds a name to a list, unless it's already there.

S328c. ⟨*printfuns.c* S328b⟩+≡ ◁S328b S328d ▷

```
static Namelist addname(Name n, Namelist xs) {
    if (nameinlist(n, xs))
        return xs;
    else
        return mkNL(n, xs);
}
```

Function `freevars` is passed an expression, a list of variables known to be bound, and a list of variables known to be free. If the expression contains free variables not on either list, `freevars` adds them to the free list and returns the new free list. Function `freevars` works by traversing an abstract-syntax tree; when it finds a name, it calls `addfree` to calculate the new list of free variables

S328d. ⟨*printfuns.c* S328b⟩+≡ ◁S328c S329a ▷

```
static Namelist addfree(Name n, Namelist bound, Namelist free) {
    if (nameinlist(n, bound))
        return free;
    else
        return addname(n, free);
}
```

The `freevars` function has to know all the rules for environments, which makes computing the free variables of an expression about as much work as evaluating the expression.

S329a. ⟨*printfuns.c* S328b⟩+≡ ◁S328d S330b▷

```
Namelist freevars(Exp e, Namelist bound, Namelist free) {
    switch (e->alt) {
    case LITERAL:
        break;
    case VAR:
        free = addfree(e->var, bound, free);
        break;
    case IFX:
        free = freevars(e->ifx.cond, bound, free);
        free = freevars(e->ifx.truex, bound, free);
        free = freevars(e->ifx.falsex, bound, free);
        break;
    case WHILEX:
        free = freevars(e->whilex.cond, bound, free);
        free = freevars(e->whilex.body, bound, free);
        break;
    case BEGIN:
        for (Explist es = e->begin; es; es = es->tl)
            free = freevars(es->hd, bound, free);
        break;
    case SET:
        free = addfree(e->set.name, bound, free);
        free = freevars(e->set.exp, bound, free);
        break;
    case APPLY:
        free = freevars(e->apply.fn, bound, free);
        for (Explist es = e->apply.actuals; es; es = es->tl)
            free = freevars(es->hd, bound, free);
        break;
    case LAMBDAX:
        ⟨let free be the free variables for e->lambdax S329b⟩
        break;
    case LETX:
        ⟨let free be the free variables for e->letx S330a⟩
        break;
    ⟨extra cases for finding free variables in μScheme expressions S340b⟩
    }
    return free;
}
```

type Exp	𝒜
type Explist	S309b
mkNil	𝒜
mkNL	𝒜
mkNum	𝒜
mkPrimitive	𝒜
mkSym	𝒜
type Name	43a
type Namelist	43a
strtoname	43b
truev	S327b
unary	S312b
type Value	𝒜

The interesting case is the one for lambda expressions. Any variables that are bound by the `lambda` are added to the "known bound" list for the recursive examination of the `lambda`'s body.

S329b. ⟨*let* `free` *be the free variables for* `e->lambdax` S329b⟩≡ (S329a)

```
for (Namelist xs = e->lambdax.formals; xs; xs = xs->tl)
    bound = addname(xs->hd, bound);
free = freevars(e->lambdax.body, bound, free);
```

The let expressions are a bit tricky; the code must follow the rules exactly.

S330a. ⟨*let* `free` *be the free variables for* `e->letx` S330a⟩≡ (S329a)

```
switch (e->letx.let) {
    Namelist xs;   // used to visit every bound name
    Explist  es;   // used to visit every expression that is bound
case LET:
    for (es = e->letx.es; es; es = es->tl)
        free = freevars(es->hd, bound, free);
    for (xs = e->letx.xs; xs; xs = xs->tl)
        bound = addname(xs->hd, bound);
    free = freevars(e->letx.body, bound, free);
    break;
case LETSTAR:
    for (xs = e->letx.xs, es = e->letx.es
       ; xs && es
       ; xs = xs->tl, es = es->tl
       )
    {
        free  = freevars(es->hd, bound, free);
        bound = addname(xs->hd, bound);
    }
    free = freevars(e->letx.body, bound, free);
    break;
case LETREC:
    for (xs = e->letx.xs; xs; xs = xs->tl)
        bound = addname(xs->hd, bound);
    for (es = e->letx.es; es; es = es->tl)
        free = freevars(es->hd, bound, free);
    free = freevars(e->letx.body, bound, free);
    break;
}
```

Printing closures and other values

Free variables are used to print closures. A closure is printed by printing its `lambda` expression, plus the values of the free variables that are not global. (Every closure contains a zillion bindings to global variables like `cons`, `car`, `+`, and so on. If these bindings were printed, they would overwhelm the ones we actually care about.) The hard work is done by function `printnonglobals`.

A recursive μScheme function is represented by a closure whose environment includes a pointer back to the recursive function itself. If such a closure were printed by printing the values of *all* its free variables, the printer could loop forever. Instead, the looping is cut off by a `depth` parameter: when `depth` reaches 0, a closure is printed simply as `<function>`.

S330b. ⟨*printfuns.c* S328b⟩+≡ ◁S329a S331a▷

```
static void printnonglobals(Printbuf output, Namelist xs, Env env, int depth);

static void printclosureat(Printbuf output, Lambda lambda, Env env, int depth) {
    if (depth > 0) {
        Namelist vars = freevars(lambda.body, lambda.formals, NULL);
        bprint(output, "<%\\, {", lambda);
        printnonglobals(output, vars, env, depth - 1);
        bprint(output, "}>");
    } else {
        bprint(output, "<function>");
    }
}
```

That same depth parameter is also used in the value-printing functions.

S331a. ⟨*printfuns.c* S328b⟩+≡ ◁S330b S331b▷

```
static void printvalueat(Printbuf output, Value v, int depth);
⟨helper functions for printvalue S332a⟩
static void printvalueat(Printbuf output, Value v, int depth) {
    switch (v.alt){
    case NIL:
        bprint(output, "()");
        return;
    case BOOLV:
        bprint(output, v.boolv ? "#t" : "#f");
        return;
    case NUM:
        bprint(output, "%d", v.num);
        return;
    case SYM:
        bprint(output, "%n", v.sym);
        return;
    case PRIMITIVE:
        bprint(output, "<function>");
        return;
    case PAIR:
        bprint(output, "(");
        printvalueat(output, *v.pair.car, depth);
        printtail(output, *v.pair.cdr, depth);
        return;
    case CLOSURE:
        printclosureat(output, v.closure.lambda, v.closure.env, depth);
        return;
    default:
        bprint(output, "<unknown v.alt=%d>", v.alt);
        return;
    }
}
```

When a value is printed using the %v conversion specifier, the default depth is 0. That is, by default the interpreter doesn't print closures. If you need to debug, increase the default depth.

S331b. ⟨*printfuns.c* S328b⟩+≡ ◁S331a S332b▷

```
void printvalue(Printbuf output, va_list_box *box) {
    printvalueat(output, va_arg(box->ap, Value), 0);
}
```

addname	S328c
bound	S329a
bprint	S176d
type Env	153a
type Explist	S309b
free	S329a
type Lambda	𝒜
type Namelist	43a
type Printbuf	S174a
printtail	S332a
type va_list_box	S177b
type Value	𝒜

The way a PAIR is printed depends on whether the pair is a proper list, in which the cdr is a list of values, or an arbitrary cons cell, in which the cdr can be any value. The difference is handled by function printtail. If a cons cell doesn't have another cons cell or NIL in its cdr field, the car and cdr are separated by a dot. This notation has been used since McCarthy's original Lisp.

S332a. ⟨*helper functions for* printvalue S332a⟩≡ (S331a)

```
static void printtail(Printbuf output, Value v, int depth) {
    switch (v.alt) {
    case NIL:
        bprint(output, ")");
        break;
    case PAIR:
        bprint(output, " ");
        printvalueat(output, *v.pair.car, depth);
        printtail(output, *v.pair.cdr, depth);
        break;
    default:
        bprint(output, " . ");
        printvalueat(output, v, depth);
        bprint(output, ")");
        break;
    }
}
```

Finally, the implementation of printnonglobals.

S332b. ⟨*printfuns.c* S328b⟩+≡ ◁S331b S337c▷

```
Env *globalenv;
static void printnonglobals(Printbuf output, Namelist xs, Env env, int depth) {
    char *prefix = "";
    for (; xs; xs = xs->tl) {
        Value *loc = find(xs->hd, env);
        if (loc && (globalenv == NULL || find(xs->hd, *globalenv) != loc)) {
            bprint(output, "%s%n -> ", prefix, xs->hd);
            prefix = ", ";
            printvalueat(output, *loc, depth);
        }
    }
}
```

L.7 μSCHEME'S UNIT TESTS

Running a list of unit tests is the job of the function process_tests. It's just like the process_tests for Impcore in Section K.2.4, except that instead of Impcore's separate function and value environments, the μScheme version uses the single μScheme environment.

S332c. ⟨*scheme-tests.c* S332c⟩≡ S333b▷

```
void process_tests(UnitTestlist tests, Env rho) {
    set_error_mode(TESTING);
    int npassed = number_of_good_tests(tests, rho);
    set_error_mode(NORMAL);
    int ntests  = lengthUL(tests);
    report_test_results(npassed, ntests);
}
```

Function number_of_good_tests runs each test, last test first, and counts the number that pass. So it can catch errors during testing, it expects the error mode to be TESTING; calling number_of_good_tests when the error mode is NORMAL is an *unchecked* run-time error. Again, except for the environment, it's just like the Impcore version.

S333a. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S325c S333c▷

```
int number_of_good_tests(UnitTestlist tests, Env rho);
```

S333b. ⟨*scheme-tests.c* S332c⟩+≡ ◁S332c S333d▷

```
int number_of_good_tests(UnitTestlist tests, Env rho) {
    if (tests == NULL)
        return 0;
    else {
        int n = number_of_good_tests(tests->tl, rho);
        switch (test_result(tests->hd, rho)) {
        case TEST_PASSED: return n+1;
        case TEST_FAILED: return n;
        default:          assert(0);
        }
    }
}
```

And except for the environment, `test_result` is just like the Impcore version.

S333c. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S333a S335g▷

```
TestResult test_result(UnitTest t, Env rho);
```

S333d. ⟨*scheme-tests.c* S332c⟩+≡ ◁S333b S335h▷

```
TestResult test_result(UnitTest t, Env rho) {
    switch (t->alt) {
    case CHECK_EXPECT:
        ⟨run check-expect test t, returning TestResult S334a⟩
    case CHECK_ASSERT:
        ⟨run check-assert test t, returning TestResult S334b⟩
    case CHECK_ERROR:
        ⟨run check-error test t, returning TestResult S334c⟩
    default:
        assert(0);
    }
}
```

Aside from the environment, μScheme's `check-expect` differs from Impcore's `check-expect` in two other ways.

- In Impcore, values are integers, they are tested for equality using C's `!=` operator. In μScheme, values are S-expressions, and they are tested for equality using C function `equalpairs` (defined below), which works the same way as the μScheme function `equal?`.

- Before being evaluated, every expression under test is passed to function `testexp`. In μScheme, this function does nothing, but in μScheme+ (Chapter 3), the function *lowers* the expression, which translates away syntactic sugar.

S334a. ⟨*run* `check-expect` *test* `t`*, returning* `TestResult` S334a⟩≡ (S333d)

```
{   if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_expect.check failed with an error S335b⟩
        bufreset(errorbuf);
        return TEST_FAILED;
    }
    Value check = eval(testexp(t->check_expect.check), rho);
    if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_expect.expect failed with an error S335c⟩
        bufreset(errorbuf);
        return TEST_FAILED;
    }
    Value expect = eval(testexp(t->check_expect.expect), rho);

    if (!equalpairs(check, expect)) {
        ⟨report failure because the values are not equal S335a⟩
        return TEST_FAILED;
    } else {
        return TEST_PASSED;
    }
}
```

And `check-assert` is like Impcore, except the test for truth is different: only a Boolean `#f` represents falsehood.

S334b. ⟨*run* `check-assert` *test* `t`*, returning* `TestResult` S334b⟩≡ (S333d)

```
{   if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_assert failed with an error S335e⟩
        bufreset(errorbuf);
        return TEST_FAILED;
    }
    Value v = eval(testexp(t->check_assert), rho);

    if (v.alt == BOOLV && !v.boolv) {
        ⟨report failure because the value is false S335d⟩
        return TEST_FAILED;
    } else {
        return TEST_PASSED;
    }
}
```

A `check-error` needn't test for equality, so again, except for the environment, it is just as in Impcore.

S334c. ⟨*run* `check-error` *test* `t`*, returning* `TestResult` S334c⟩≡ (S333d)

```
{   if (setjmp(testjmp)) {
        bufreset(errorbuf);
        return TEST_PASSED; // error occurred, so the test passed
    }
    Value check = eval(testexp(t->check_error),  rho);
    ⟨report that evaluating t->check_error produced check S335f⟩
    return TEST_FAILED;
}
```

And test results are reported as in Impcore.

S335a. ⟨*report failure because the values are not equal* S335a⟩≡ (S334a)

```
fprint(stderr, "Check-expect failed: expected %e to evaluate to %v",
       t->check_expect.check, expect);
if (t->check_expect.expect->alt != LITERAL)
    fprint(stderr, " (from evaluating %e)", t->check_expect.expect);
fprint(stderr, ", but it's %v.\n", check);
```

S335b. ⟨*report that evaluating* `t->check_expect.check` *failed with an error* S335b⟩≡ (S334a)

```
fprint(stderr, "Check-expect failed: expected %e to evaluate to the same "
               "value as %e, but evaluating %e causes an error: %s.\n",
               t->check_expect.check, t->check_expect.expect,
               t->check_expect.check, bufcopy(errorbuf));
```

S335c. ⟨*report that evaluating* `t->check_expect.expect` *failed with an error* S335c⟩≡ (S334a)

```
fprint(stderr, "Check-expect failed: expected %e to evaluate to the same "
               "value as %e, but evaluating %e causes an error: %s.\n",
               t->check_expect.check, t->check_expect.expect,
               t->check_expect.expect, bufcopy(errorbuf));
```

S335d. ⟨*report failure because the value is false* S335d⟩≡ (S334b)

```
fprint(stderr, "Check-assert failed: %e evaluates to #f.\n", t->check_assert);
```

S335e. ⟨*report that evaluating* `t->check_assert` *failed with an error* S335e⟩≡ (S334b)

```
fprint(stderr, "Check-assert failed: evaluating %e causes an error: %s.\n",
               t->check_assert, bufcopy(errorbuf));
```

S335f. ⟨*report that evaluating* `t->check_error` *produced* `check` S335f⟩≡ (S334c)

```
fprint(stderr, "Check-error failed: evaluating %e was expected to produce "
               "an error, but instead it produced the value %v.\n",
               t->check_error, check);
```

The equality check in `check-expect` is implemented by function `equalpairs`. It resembles function `equalatoms` (chunk S314a), which implements the primitive =, with two differences:

- Its semantics are those of `equal?`, not =.
- Instead of returning a μScheme Boolean represented as a C `Value`, it returns a Boolean represented as a C `bool`.

S335g. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S333c S336a▷

```
bool equalpairs(Value v, Value w);
```

S335h. ⟨*scheme-tests.c* S332c⟩+≡ ◁S333d

```
bool equalpairs(Value v, Value w) {
    if (v.alt != w.alt)
        return false;
    else
        switch (v.alt) {
        case PAIR:
            return equalpairs(*v.pair.car, *w.pair.car) &&
                   equalpairs(*v.pair.cdr, *w.pair.cdr);
        case NUM:
            return v.num   == w.num;
        case BOOLV:
            return v.boolv == w.boolv;
        case SYM:
            return v.sym   == w.sym;
        case NIL:
            return true;
        default:
            return false;
        }
}
```

`bufcopy`	S174d
`bufreset`	S174c
`errorbuf`	S182
`eval`	155a
`fprint`	S176d
`rho`	S333d
`testexp`	S336a
`testjmp`	S181b
`type Value`	$\mathcal{A}$

In *μ*Scheme, expressions under test don't have to be lowered: `testexp` returns its argument unchanged.

S336a. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S335g S337a▷

```
Exp testexp(Exp);
```

S336b. ⟨*eval.c* S336b⟩≡

```
Exp testexp(Exp e) {
    return e;
}
```

L.8 Parse-time error checking

*μ*Scheme requires that the names of formal parameters and `let`-bound variables be mutually distinct. If the requirement isn't met, a syntax error is signaled at parse time by function `check_exp_duplicates`. This function also checks the requirement that every right-hand side in a `letrec` expression is a `lambda`.

S336c. ⟨*parse.c* S322b⟩+≡ ◁S327a S336d▷

```
void check_exp_duplicates(Sourceloc source, Exp e) {
    switch (e->alt) {
    case LAMBDAX:
        if (duplicatename(e->lambdax.formals) != NULL)
            synerror(source, "formal parameter %n appears twice in lambda",
                     duplicatename(e->lambdax.formals));
        return;
    case LETX:
        if (e->letx.let != LETSTAR && duplicatename(e->letx.xs) != NULL)
            synerror(source, "bound name %n appears twice in %s",
                     duplicatename(e->letx.xs),
                     e->letx.let == LET ? "let" : "letrec");
        if (e->letx.let == LETREC)
            for (Explist es = e->letx.es; es; es = es->tl)
                if (es->hd->alt != LAMBDAX)
                    synerror(source,
                             "in letrec, expression %e is not a lambda", es->hd);
        return;
    default:
        return;
    }
}
```

Each `define` form also has to be checked.

S336d. ⟨*parse.c* S322b⟩+≡ ◁S336c S337b▷

```
void check_def_duplicates(Sourceloc source, Def d) {
    if (d->alt == DEFINE && duplicatename(d->define.lambda.formals) != NULL)
        synerror(source,
                 "formal parameter %n appears twice in define",
                 duplicatename(d->define.lambda.formals));
}
```

L.9 Support for an exercise: Concatenating names

If you implement μScheme's syntactic sugar for records (Exercise 54 on page 196), you will find a use for function `namecat`, which concatenates names. (The names of record-accessor and record-constructor functions are formed by concatenation.)

S337a. ⟨*function prototypes for μScheme* S309d⟩+≡ (S318a) ◁S336a

```
Name namecat(Name n1, Name n2);
```

S337b. ⟨*parse.c* S322b⟩+≡ ◁S336d

```
Name namecat(Name n1, Name n2) {
    const char *s1 = nametostr(n1);
    const char *s2 = nametostr(n2);
    char *buf = malloc(strlen(s1) + strlen(s2) + 1);
    assert(buf);
    sprintf(buf, "%s%s", s1, s2);
    Name answer = strtoname(buf);
    free(buf);
    return answer;
}
```

L.10 Print functions for expressions

μScheme defines special conversion specifiers for printing abstract-syntax trees. The (boring) code the does the printing is written here.

S337c. ⟨*printfuns.c* S328b⟩+≡ ◁S332b S338a▷

```
void printdef(Printbuf output, va_list_box *box) {
    Def d = va_arg(box->ap, Def);
    if (d == NULL) {
        bprint(output, "<null>");
        return;
    }

    switch (d->alt) {
    case VAL:
        bprint(output, "(val %n %e)", d->val.name, d->val.exp);
        return;
    case EXP:
        bprint(output, "%e", d->exp);
        return;
    case DEFINE:
        bprint(output, "(define %n %\\)", d->define.name, d->define.lambda);
        return;
    }
    assert(0);
}
```

`bprint`	S176d
type `Def`	$\mathcal{A}$
`duplicatename`	S184d
type `Exp`	$\mathcal{A}$
type `Explist`	S309b
type `Name`	43a
`nametostr`	43b
type `Printbuf`	S174a
type `Sourceloc`	S293h
`strtoname`	43b
`synerror`	47b
type `va_list_box`	S177b

S338a. ⟨*printfuns.c* S328b⟩+≡ ◁S337c S338b▷

```
void printxdef(Printbuf output, va_list_box *box) {
    XDef d = va_arg(box->ap, XDef);
    if (d == NULL) {
        bprint(output, "<null>");
        return;
    }

    switch (d->alt) {
    case USE:
        bprint(output, "(use %n)", d->use);
        return;
    case TEST:
        bprint(output, "CANNOT PRINT UNIT TEST XXX\n");
        return;
    case DEF:
        bprint(output, "%t", d->def);
        return;
    }
    assert(0);
}
```

S338b. ⟨*printfuns.c* S328b⟩+≡ ◁S338a S339a▷

```
static void printlet(Printbuf output, Exp let) {
    switch (let->letx.let) {
    case LET:
        bprint(output, "(let (");
        break;
    case LETSTAR:
        bprint(output, "(let* (");
        break;
    case LETREC:
        bprint(output, "(letrec (");
        break;
    default:
        assert(0);
    }
    Namelist xs;  // visits every let-bound name
    Explist es;   // visits every bound expression
    for (xs = let->letx.xs, es = let->letx.es;
         xs && es;
         xs = xs->tl, es = es->tl)
        bprint(output, "(%n %e)%s", xs->hd, es->hd, xs->tl?" ":"");
    bprint(output, ") %e)", let->letx.body);
}
```

S339a. ⟨*printfuns.c* S328b⟩+≡ ◁S338b S339b▷

```
void printexp(Printbuf output, va_list_box *box) {
    Exp e = va_arg(box->ap, Exp);
    if (e == NULL) {
        bprint(output, "<null>");
        return;
    }

    switch (e->alt) {
    case LITERAL:
        if (e->literal.alt == NUM || e->literal.alt == BOOLV)
            bprint(output, "%v", e->literal);
        else
            bprint(output, "'%v", e->literal);
        break;
    case VAR:
        bprint(output, "%n", e->var);
        break;
    case IFX:
        bprint(output, "(if %e %e %e)", e->ifx.cond, e->ifx.truex, e->ifx.falsex);
        break;
    case WHILEX:
        bprint(output, "(while %e %e)", e->whilex.cond, e->whilex.body);
        break;
    case BEGIN:
        bprint(output, "(begin%s%E)", e->begin ? " " : "", e->begin);
        break;
    case SET:
        bprint(output, "(set %n %e)", e->set.name, e->set.exp);
        break;
    case LETX:
        printlet(output, e);
        break;
    case LAMBDAX:
        bprint(output, "%\\", e->lambdax);
        break;
    case APPLY:
        bprint(output, "(%e%s%E)", e->apply.fn,
              e->apply.actuals ? " " : "", e->apply.actuals);
        break;
    ⟨extra cases for printing μScheme ASTs S340a⟩
    default:
        assert(0);
    }
}
```

S339b. ⟨*printfuns.c* S328b⟩+≡ ◁S339a

```
void printlambda(Printbuf output, va_list_box *box) {
    Lambda l = va_arg(box->ap, Lambda);
    bprint(output, "(lambda (%N) %e)", l.formals, l.body);
}
```

`bprint`	S176d
type `Exp`	$\mathcal{A}$
type `Explist`	S309b
type `Lambda`	$\mathcal{A}$
type `Namelist`	43a
type `Printbuf`	S174a
type `va_list_box`	S177b
type `XDef`	$\mathcal{A}$

L.11 Support for μScheme+

In three locations, the μScheme interpreter contains empty placeholders which, in Chapter 3, are filled with code that implements parts of μScheme+. (μScheme+ is an extension that adds control operators to μScheme.) The placeholders are defined as follows:

S340a. ⟨*extra cases for printing μScheme ASTs* S340a⟩≡ (S339a)

S340b. ⟨*extra cases for finding free variables in μScheme expressions* S340b⟩≡ (S329a)

S340c. ⟨*lowering functions for μScheme+* S340c⟩≡ (S323a)

```
/* placeholder */
```

L.12 Placeholder for desugaring

As described in Chapter 2 (page 163), every `let` form can be desugared into a combination of `lambda` expressions and function applications. The chapter shows code for desugaring `let*`, but if you want to desugar `let`, you'll need to write C code to replace this function:

S340d. ⟨*parse.c* **[[prototype]]** S340d⟩≡

```
Exp desugarLet(Namelist xs, Explist es, Exp body) {
    /* you replace the body of this function */
    runerror("desugaring for LET never got implemented");
    return NULL;
}
```

Appendix M contents

M Supporting material for μScheme+

This appendix presents all the C code that didn't make sense to show in Chapter 3. It also includes a deleted scene and a couple of bonus exercises.

M.1 The evaluation stack

This section shows the implementation of the `Stack` of evaluation contexts and its instrumentation.

M.1.1 Implementing the stack

In Chapter 3, the representation of a `Stack` is private to this module. In Chapter 4, the representation is exposed to the garbage collector.

S343a. ⟨*representation of* `struct Stack` S343a⟩≡ (S343b)

```
struct Stack {
    int size;
    Frame *frames;  // memory for 'size' frames
    Frame *sp;      // points to first unused frame
};
```

The stack grows toward larger addresses, so a stack `s` satisfies these invariants:

- The number of frames on the stack is `s->sp - s->frames`.
- `s->frames` $\leq$ `s->sp` $\leq$ `s->frames` $+$ `s->size`.

Instrumentation is stored in three global variables. Tail-call optimization is on by default; showing the high stack mark is not.

S343b. ⟨*context-stack.c* S343b⟩≡ S343c ▷

```
⟨representation of struct Stack S343a⟩

bool optimize_tail_calls = true;
int  high_stack_mark;   // max number of frames used in the current evaluation
bool show_high_stack_mark;
```

A fresh, empty stack can hold 8 frames.

S343c. ⟨*context-stack.c* S343b⟩+≡ ◁ S343b S344a ▷

```
Stack emptystack(void) {
    Stack s;
    s = malloc(sizeof *s);
    assert(s);
    s->size = 8;
    s->frames = malloc(s->size * sizeof(*s->frames));
    assert(s->frames);
    s->sp = s->frames;
    return s;
}
```

A stack that has already been allocated can be emptied by calling `clearstack`. For example, `evalstack` may be emptied if a call to `eval` is terminated prematurely (with a nonempty stack) by a call to `error`.

S344a. ⟨*context-stack.c* S343b⟩+≡ ◁S343c S344c▷

```
void clearstack (Stack s) {
    s->sp = s->frames;
}
```

On every evaluation, the `evalstack` structure is initialized or reset.

S344b. ⟨*ensure that* `evalstack` *is initialized and empty* S344b⟩≡ (227a)

```
if (evalstack == NULL)
    evalstack = emptystack();
else
    clearstack(evalstack);
```

The frame on top of the stack (that is, the young end) is returned by `topframe`. A top frame is present unless the `sp` and `frames` fields point to the same memory.

S344c. ⟨*context-stack.c* S343b⟩+≡ ◁S344a S344d▷

```
Frame *topframe (Stack s) {
    assert(s);
    if (s->sp == s->frames)
        return NULL;
    else
        return s->sp - 1;
}
```

To implement tail-call optimization when `return` has been lowered (Exercise 23 in Chapter 3), you will need to find the frame nearest the top (the youngest frame) that is not a label.

S344d. ⟨*context-stack.c* S343b⟩+≡ ◁S344c S344e▷

```
Frame *topnonlabel (Stack s) {
    Frame *p;
    for (p = s->sp; p > s->frames && p[-1].form.alt == LONG_LABEL; p--)
        ;
    if (p > s->frames)
        return p-1;
    else
        return NULL;
}
```

A frame is pushed, whether by `pushframe` or `pushenv_opt`, using the private function `push`. Function `push` returns a pointer to the frame just pushed.

S344e. ⟨*context-stack.c* S343b⟩+≡ ◁S344d S345b▷

```
static Frame *push (Frame f, Stack s) {
    assert(s);
    ⟨if stack s is full, enlarge it S345a⟩
    *s->sp++ = f;
    ⟨set high_stack_mark from stack s S346f⟩
    return s->sp - 1;
}
```

Pushing a frame must not cause a stack to grow without bound. Ten thousand stack frames ought to be enough for anybody.

S345a. ⟨*if stack* s *is full, enlarge it* S345a⟩≡ (S344e)

```
if (s->sp - s->frames == s->size) {
    unsigned newsize = 2 * s->size;
    if (newsize > 10000) {
        clearstack(s);
        runerror("recursion too deep");
    }
    s->frames = realloc(s->frames, newsize * sizeof(*s->frames));
    assert(s->frames);
    s->sp = s->frames + s->size;
    s->size = newsize;
}
```

Code in Chapter 3 is simplified by using pushframe, which pushes syntax.

S345b. ⟨*context-stack.c* S343b⟩+≡ ◁ S344e S345c ▷

```
static Frame mkExpFrame(struct Exp e) {
  Frame fr;
  fr.form = e;
  fr.syntax = NULL;
  return fr;
}

Exp pushframe(struct Exp e, Stack s) {
  Frame *fr;
  assert(s);
  fr = push(mkExpFrame(e), s);
  return &fr->form;
}
```

Any frame that is pushed can be popped. But since popping doesn't require allocating memory or updating the high stack mark, it doesn't require memory management or instrumentation.

S345c. ⟨*context-stack.c* S343b⟩+≡ ◁ S345b S346a ▷

```
void popframe (Stack s) {
    assert(s->sp - s->frames > 0);
    s->sp--;
}
```

M.1.2 *Printing the stack*

Frames, stacks, and environments are printed by conversion specifiers %F, %S, and %R, which are implemented here. Function printnoenv prints the current environment as a C pointer, not as a list of (name, value) pairs.

S345d. ⟨*function prototypes for μScheme+* S345d⟩≡ (S358) S351b ▷

```
void printframe   (Printbuf, Frame *fr);
void printoneframe(Printbuf, va_list_box*);
void printstack   (Printbuf, va_list_box*);
void printnoenv   (Printbuf, va_list_box*);
```

S345e. ⟨*install printers* S345e⟩≡

```
installprinter('F', printoneframe);
installprinter('S', printstack);
installprinter('R', printnoenv);
```

S346a. ⟨*context-stack.c* S343b⟩+≡ ◁ S345c S346b ▷

```
void printframe (Printbuf output, Frame *fr) {
    bprint(output, "%*: ", (void *) fr);
    bprint(output, "[%e]", &fr->form);
}
```

S346b. ⟨*context-stack.c* S343b⟩+≡ ◁ S346a S346c ▷

```
void printoneframe(Printbuf output, va_list_box *box) {
    Frame *fr = va_arg(box->ap, Frame*);
    printframe(output, fr);
}
```

S346c. ⟨*context-stack.c* S343b⟩+≡ ◁ S346b S346d ▷

```
void printstack(Printbuf output, va_list_box *box) {
    Stack s = va_arg(box->ap, Stack);
    Frame *fr;

    for (fr = s->sp-1; fr >= s->frames; fr--) {
        bprint(output, "  ");
        printframe(output, fr);
        bprint(output, ";\n");
    }
}
```

S346d. ⟨*context-stack.c* S343b⟩+≡ ◁ S346c

```
void printnoenv(Printbuf output, va_list_box* box) {
    Env env = va_arg(box->ap, Env);
    bprint(output, "@%*", (void *)env);
}
```

M.1.3 Instrumentation for the high stack mark

During an evaluation, the maximum size reached by the stack is called the *high stack mark*. It is tracked here

S346e. ⟨*use the options in* `env` *to initialize the instrumentation* S346e⟩≡ (227a) S347c ▷

```
high_stack_mark = 0;
show_high_stack_mark =
    istrue(getoption(strtoname("&show-high-stack-mark"), env, falsev));
```

S346f. ⟨*set* `high_stack_mark` *from stack* `s` S346f⟩≡ (S344e)

```
{   int n = s->sp - s->frames;
    if (n > high_stack_mark)
        high_stack_mark = n;
}
```

At the end of an evaluation, the high stack mark can be recorded.

S346g. ⟨*if* `show_high_stack_mark` *is set, show maximum stack size* S346g⟩≡ (227b)

```
if (show_high_stack_mark)
    fprintf(stderr, "High stack mark == %d\n", high_stack_mark);
```

M.1.4 Tracing machine state using the stack

The state of the stack can be displayed at every step of an evaluation. The steps are identified by variables `etick` and `vtick`, which count the number of state transitions involving an expression or a variable as the current item, respectively.

S346h. ⟨*stack-debug.c* S346h⟩≡ S347a ▷

```
static int etick, vtick;  // number of times saw a current expression or value
```

Tracing is controlled by the value of the μScheme+ variable `&trace-stack`. Placing control in a μScheme+ variable enables μScheme+ code to turn tracing on and off during a single call to `eval`. If the value of `&trace-stack` is a number, then pointer `trace_countp` points to that value.

S347a. ⟨*stack-debug.c* S346h⟩+≡ ◁S346h S347b▷

```
static int *trace_countp; // if not NULL, points to value of &trace-stack
```

All three variables are initialized by function `stack_trace_init`, which receives a pointer to `&trace-stack` if it exists and is a number.

S347b. ⟨*stack-debug.c* S346h⟩+≡ ◁S347a S347d▷

```
void stack_trace_init(int *countp) {
    etick = vtick = 0;
    trace_countp = countp;
}
```

Function `stack_trace_init` is called from `eval`, which has access to `env`. The call does just a little sanity checking. The sanity check does not prevent μScheme+ code from changing the value of `&trace-stack` to a non-number. If that happens, chaos may ensue.

S347c. ⟨*use the options in* `env` *to initialize the instrumentation* S346e⟩+≡ (227a) ◁S346e S356d▷

```
{   Value *p = find(strtoname("&trace-stack"), env);
    if (p && p->alt == NUM)
        stack_trace_init(&p->num);
    else
        stack_trace_init(NULL);
}
```

The `etick` number is shown when tracing a current expression. So are the expression, a pointer to the environment, and the stack. And every time a trace is shown, the trace count is decremented.

S347d. ⟨*stack-debug.c* S346h⟩+≡ ◁S347b S347e▷

```
void stack_trace_current_expression(Exp e, Env rho, Stack s) {
    if (trace_countp && *trace_countp != 0) {
        (*trace_countp)--;
        etick++;
        fprint(stderr, "exp  %d = %e\n", etick, e);
        fprint(stderr, "env  %R\n", rho);
        fprint(stderr, "stack\n%S\n", s);
    }
}
```

When tracing a current value, the `vtick` number is used in the same way. And the empty stack is rendered to show that when there is a current value and an empty stack, evaluation is complete.

S347e. ⟨*stack-debug.c* S346h⟩+≡ ◁S347d

```
void stack_trace_current_value(Value v, Env rho, Stack s) {
    if (trace_countp && *trace_countp != 0) {
        (*trace_countp)--;
        vtick++;
        fprint(stderr, "val  %d = %v\n", vtick, v);
        fprint(stderr, "env  %R\n", rho);
        if (topframe(s))
            fprint(stderr, "stack\n%S\n", s);
        else
            fprint(stderr, " (final answer from stack-based eval)\n");
    }
}
```

M.2 Updating lists of expressions within contexts

Evaluation rules for APPLY, LET, and other forms can call for the evaluator to update an expression or a hole within an `Explist`. The functions used in the updates are described in Section 3.6.8 and are implemented here.

S348a. ⟨*context-lists.c* S348a⟩≡ S348b ▷

  ⟨*private functions for updating lists of expressions in contexts* S348c⟩

To move a hole from one position to the next, `transition_explist` finds the hole, fills it, and then places a new hole at the beginning of the rest of the list.

S348b. ⟨*context-lists.c* S348a⟩+≡ ◁ S348a S348e ▷

```
Exp transition_explist(Explist es, Value v) {
  Explist p = find_explist_hole(es);
  assert(p);
  fill_hole(p->hd, v);
  return head_replaced_with_hole(p->tl);
}
```

S348c. ⟨*private functions for updating lists of expressions in contexts* S348c⟩≡ (S348a) S348d ▷

```
static void fill_hole(Exp e, Value v) {
  assert(e->alt == HOLE);
  e->alt = LITERAL;
  e->literal = v;
}
```

Function `find_explist_hole` returns a pointer to the first hole in a list of expressions, or if there is no hole, returns `NULL`.

S348d. ⟨*private functions for updating lists of expressions in contexts* S348c⟩+≡ (S348a) ◁ S348c

```
static Explist find_explist_hole(Explist es) {
  while (es && es->hd->alt != HOLE)
    es = es->tl;
  return es;
}
```

Function `head_replaced_with_hole(es)` replaces the head of list `es` with a hole, returning the old head. If list `es` is empty, `head_replaced_with_hole` returns `NULL`. Function `head_replaced_with_hole` doesn't allocate space for each new result—all results share the same space.

S348e. ⟨*context-lists.c* S348a⟩+≡ ◁ S348b S349a ▷

```
Exp head_replaced_with_hole(Explist es) {
  static struct Exp a_copy; // overwritten by subsequent calls
  if (es) {
    a_copy = *es->hd;
    *es->hd = mkHoleStruct();
    return &a_copy;
  } else {
    return NULL;
  }
}
```

Function `copyEL` copies not only the `Explist` pointers but also the `Exp` pointers they hold.

S349a. ⟨*context-lists.c* S348a⟩+≡ ◁S348e S349b▷

```
Explist copyEL(Explist es) {
  if (es == NULL)
    return NULL;
  else {
    Exp e = malloc(sizeof(*e));
    assert(e);
    *e = *es->hd;
    return mkEL(e, copyEL(es->tl));
  }
}
```

Correspondingly, `freeEL` frees both the `Explist` pointers and the internal `Exp` pointers.

S349b. ⟨*context-lists.c* S348a⟩+≡ ◁S349a S349c▷

```
void freeEL(Explist es) {
  if (es != NULL) {
    freeEL(es->tl);
    free(es->hd);
    free(es);
  }
}
```

By contrast, a `Valuelist` contains no internal pointers, so only the `Valuelist` pointers can be freed.

S349c. ⟨*context-lists.c* S348a⟩+≡ ◁S349b S349d▷

```
void freeVL(Valuelist vs) {
  if (vs != NULL) {
    freeVL(vs->tl);
    free(vs);
  }
}
```

Conversion of an `Explist` to a `Valuelist` requires allocation and therefore incurs an obligation to call `freeVL` on the result.

S349d. ⟨*context-lists.c* S348a⟩+≡ ◁S349c S349e▷

```
Valuelist asLiterals(Explist es) {
  if (es == NULL)
    return NULL;
  else
    return mkVL(asLiteral(es->hd), asLiterals(es->tl));

}
```

By contrast, because a `Value` is not a pointer, `asLiteral` need not allocate.

S349e. ⟨*context-lists.c* S348a⟩+≡ ◁S349d

```
Value asLiteral(Exp e) {
  assert(e->alt == LITERAL);
  return validate(e->literal);
}
```

M.3 Lowering

Chapter 3 describes a *lowering* transformation that enables most control operators to be implemented by a combination of `long-goto` and `long-label`. (The lowering of `return` is left as an exercise.) The infrastructure that supports the lowering is here.

Whether to lower `return` is your choice. The choice is communicated to the compiler via this `LOWER_RETURN` macro:

S350a. ⟨*lower.c* S350a⟩≡ S350b ▷

```
#define LOWER_RETURN false // to do return-lowering exercise, change me
```

After that it's just one damn lowering function after another.

S350b. ⟨*lower.c* S350a⟩+≡ ◁ S350a S350c ▷

```
static inline Exp lowerLet1(Name x, Exp e, Exp body) {
    return mkLetx(LET, mkNL(x, NULL), mkEL(e, NULL), body);
}
```

In `lowerSequence`, the name "`ignore me`" cannot be written in the source code, so there is no danger of variable capture.

S350c. ⟨*lower.c* S350a⟩+≡ ◁ S350b S350d ▷

```
static Exp lowerSequence(Exp e1, Exp e2) {
    return lowerLet1(strtoname("ignore me"), e1, e2);
}
```

S350d. ⟨*lower.c* S350a⟩+≡ ◁ S350c S350e ▷

```
static Exp lowerBegin(Explist es) {
    if (es == NULL)
        return mkLiteral(falsev);
    else if (es->tl == NULL)
        return es->hd;
    else
        return lowerSequence(es->hd, lowerBegin(es->tl));
}
```

S350e. ⟨*lower.c* S350a⟩+≡ ◁ S350d S350f ▷

```
static Exp lower(LoweringContext c, Exp e);
static void lowerAll(LoweringContext c, Explist es) {
    if (es) {
        lowerAll(c, es->tl);
        es->hd = lower(c, es->hd);
    }
}
```

S350f. ⟨*lower.c* S350a⟩+≡ ◁ S350e S350g ▷

```
static Exp lowerLetstar(Namelist xs, Explist es, Exp body) {
    if (xs == NULL) {
        assert(es == NULL);
        return body;
    } else {
        assert(es != NULL);
        return lowerLet1(xs->hd, es->hd, lowerLetstar(xs->tl, es->tl, body));
    }
}
```

Once lowering functions for individual forms are defined, the definition of `lower` can be emitted.

S350g. ⟨*lower.c* S350a⟩+≡ ◁ S350f S351a ▷

⟨*definition of private function* `lower` 226e⟩

A definition is lowered by lowering any expressions it contains.

S351a. ⟨*lower.c* S350a⟩+≡ ◁S350g S351c▷

```
static void lowerDef(Def d) {
    switch (d->alt) {
    case VAL:    d->val.exp = lower(0, d->val.exp);     break;
    case EXP:    d->exp     = lower(0, d->exp);         break;
    case DEFINE: {
        LoweringContext c = FUNCONTEXT;
        Exp body = lower(c, d->define.lambda.body);
        if (LOWER_RETURN)
            body = mkLowered(d->define.lambda.body,
                               mkLongLabel(strtoname(":return"), body));
        d->define.lambda.body = body;
        break;
    }
    default:     assert(0);
    }
}
```

Extended definitions also have to be lowered. Extended definitions include unit tests, and a unit test can contain control operators, so unit tests have to be lowered. But a unit test can't be lowered until just before it is run—if lowering fails with a run-time error, the error has to occur in the right dynamic context.

S351b. ⟨*function prototypes for μScheme+* S345d⟩+≡ (S358) ◁S345d

```
void lowerXdef(XDef); // lower every expression in this definition
```

S351c. ⟨*lower.c* S350a⟩+≡ ◁S351a S351d▷

```
void lowerXdef(XDef d) {
    switch (d->alt) {
    case DEF: lowerDef(d->def); break;
    case USE: break;
    case TEST: break;  // lowering is delayed until testing time
    default: assert(0);
    }
}
```

A unit test is finally lowered by function `testexp`, which is called from the testing infrastructure shown in Appendix L.

S351d. ⟨*lower.c* S350a⟩+≡ ◁S351c

```
Exp testexp(Exp e) {
    return lower(0, e);
}
```

Function `lowerXdef` is called from code shown in Chapter 3.

S351e. ⟨*lower extended definition* `xd` *as needed* S351e⟩≡ (S310e)

```
lowerXdef(xd);
```

While Chapter 3 shows the *rules* for lowering all the expression forms, it shows *code* for lowering only `set` and `break`. Code for the other forms is shown here.

S351f. ⟨*other cases for lowering expression* `e` S351f⟩≡ (226e) S352a▷

```
case LITERAL: return e;
case VAR:     return e;
case IFX:     e->ifx.cond   = lower(c, e->ifx.cond);
              e->ifx.truex  = lower(c, e->ifx.truex);
              e->ifx.falsex = lower(c, e->ifx.falsex);
              return e;
```

S352a. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S351f S352b▷

```
case WHILEX: {
    LoweringContext nc = c | LOOPCONTEXT;
    Exp body = mkLongLabel(strtoname(":continue"), lower(nc, e->whilex.body));
    Exp cond = lower(c, e->whilex.cond);
    Exp placeholder   = mkLiteral(falsev); // unique pointer
    Exp loop          = mkIfx(cond, placeholder, mkLiteral(falsev));
    loop->ifx.truex = lowerSequence(body, mkLowered(e, mkLoopback(loop)));
    Exp lowered       = mkLongLabel(strtoname(":break"), loop);
    return mkLowered(e, lowered);
}
```

S352b. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S352a S352c▷

```
case BEGIN:
    lowerAll(c, e->begin);
    return mkLowered(e, lowerBegin(e->begin));
```

S352c. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S352b S352d▷

```
case LETX:
    lowerAll(c, e->letx.es);
    e->letx.body = lower(c, e->letx.body);
    switch (e->letx.let) {
    case LET: case LETREC:
        return e;
    case LETSTAR:
        return mkLowered(e, lowerLetstar(e->letx.xs, e->letx.es,
                                           e->letx.body));
    default:
        assert(0);
    }
```

S352d. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S352c S352e▷

```
case LAMBDAX: {
    LoweringContext nc = FUNCONTEXT;  // no loop!
    Exp body   = lower(nc, e->lambdax.body);
    e->lambdax.body =
        LOWER_RETURN ? mkLowered(e->lambdax.body, mkLongLabel(strtoname(":return"), body))
                     : body;
    return e;
}
```

S352e. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S352d S352f▷

```
case APPLY:
    lowerAll(c, e->apply.actuals);
    e->apply.fn = lower(c, e->apply.fn);
    return e;
```

S352f. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S352e S353a▷

```
case CONTINUEX:
    if (c & LOOPCONTEXT)
        return mkLowered(e, mkLongGoto(strtoname(":continue"), mkLiteral(falsev)));
    else
        othererror("Lowering error: %e appeared outside of any loop", e);
case RETURNX:
    e->returnx = lower(c, e->returnx);
    if (c & FUNCONTEXT)
        return LOWER_RETURN ? mkLowered(e, mkLongGoto(strtoname(":return"), e->returnx))
                            : e;
    else
        othererror("Lowering error: %e appeared outside of any function", e);
```

S353a. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S352f S353b▷

```
case TRY_CATCH: {
    Exp body    = lower(c, e->try_catch.body);
    Exp handler = lower(c, e->try_catch.handler);
    Name h = strtoname("the-;-handler");
    Name x = strtoname("the-;-answer");
    Exp lbody = lowerLet1(x, body,
                      mkLambdax(mkLambda(mkNL(strtoname("_"), NULL),
                                                   mkVar(x))));
    Exp labeled = mkLongLabel(e->try_catch.label, lbody);
    Exp lowered = lowerLet1(h, handler, mkApply(labeled, mkEL(mkVar(h), NULL)));
    return mkLowered(e, lowered);
}
```

S353b. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S353a S353c▷

```
case THROW: {
    Name h = strtoname("the-;-handler");
    Name x = strtoname("the-;-answer");
    Lambda thrown =
        mkLambda(mkNL(h, NULL), mkApply(mkVar(h), mkEL(mkVar(x), NULL)));
    Exp throw = mkLongGoto(e->throw.label, mkLambdax(thrown));
    Exp lowered = lowerLet1(x, lower(c, e->throw.exp), throw);
    return mkLowered(e, lowered);
}
```

S353c. ⟨*other cases for lowering expression* e S351f⟩+≡ (226e) ◁S353b

```
case LONG_LABEL:
    e->long_label.body = lower(c, e->long_label.body);
    return e;
case LONG_GOTO:
    e->long_goto.exp = lower(c, e->long_goto.exp);
    return e;
case LOWERED: case LOOPBACK:
    assert(0); // never expect to lower twice
default:
    assert(0);
```

M.4 Parsing

μScheme+ extends μScheme with new syntactic forms, which means extending the parsing tables and functions shown in Appendix L. The new forms are treated as if they were added in exercises.

S353d. ⟨*arrays of shift functions added to μScheme in exercises* S353d⟩≡ (S323a) S354a▷

```
ShiftFun breakshifts[]  = { stop };
ShiftFun returnshifts[] = { sExp, stop };
ShiftFun throwshifts[]  = { sName, sExp, stop };
ShiftFun tcshifts[]     = { sExp, sName, sExp, stop };
ShiftFun labelshifts[]  = { sName, sExp, stop };
```

S353e. ⟨*rows added to μScheme's* exptable *in exercises* S353e⟩≡ (S323a)

```
{ "break",      BREAKX,        breakshifts },
{ "continue",   CONTINUEX,     breakshifts },
{ "return",     RETURNX,       returnshifts },
{ "throw",      THROW,         throwshifts },
{ "try-catch",  TRY_CATCH,     tcshifts },
{ "long-label", LONG_LABEL,    labelshifts },
{ "long-goto",  LONG_GOTO,     labelshifts },
{ "when",       SUGAR(WHEN),   applyshifts },
{ "unless",     SUGAR(UNLESS), applyshifts },
```

S354a. ⟨*arrays of shift functions added to μScheme in exercises* S353d⟩+≡ (S323a) ◁ S353d

```
static ShiftFun procwhileshifts[] = { sExp, sExps,      stop };
static ShiftFun procletshifts[]   = { sBindings, sExps, stop };
```

OK, not really exercises…

S354b. ⟨*rows of μScheme's* `exptable` *that are sugared in μScheme+* **⟦uschemeplus⟧** S354b⟩≡

```
  { "while",  ANEXP(WHILEX),  procwhileshifts },
  { "let",    ALET(LET),      procletshifts },
  { "let*",   ALET(LETSTAR),  procletshifts },
  { "letrec", ALET(LETREC),   procletshifts },
```

S354c. ⟨*cases for μScheme's* `reduce_to_exp` *added in exercises* S354c⟩≡ (S323d)

```
case ANEXP(BREAKX):     return mkBreakx();
case ANEXP(CONTINUEX):  return mkContinuex();
case ANEXP(RETURNX):    return mkReturnx(comps[0].exp);
case ANEXP(THROW):      return mkThrow(comps[0].name, comps[1].exp);
case ANEXP(TRY_CATCH):  return mkTryCatch(comps[0].exp,
                                          comps[1].name, comps[2].exp);
case ANEXP(LONG_LABEL): return mkLongLabel(comps[0].name, comps[1].exp);
case ANEXP(LONG_GOTO):  return mkLongGoto(comps[0].name, comps[1].exp);
case SUGAR(WHEN):       return mkIfx(comps[0].exp, smartBegin(comps[1].exps),
                                     mkLiteral(falsev));
case SUGAR(UNLESS):     return mkIfx(comps[0].exp, mkLiteral(falsev),
                                     smartBegin(comps[1].exps));
```

S354d. ⟨*cases for* `reduce_to_exp` *that are sugared in μScheme+* **⟦uschemeplus⟧** S354d⟩≡

```
case ANEXP(WHILEX):  (void) whileshifts;
                     return mkWhilex(comps[0].exp, smartBegin(comps[1].exps));
case ALET(LET):
case ALET(LETSTAR):
case ALET(LETREC):   (void) letshifts;
                     return mkLetx(code+LET-ALET(LET),
                                   comps[0].names, comps[0].exps,
                                   smartBegin(comps[1].exps));
```

S354e. ⟨*μScheme* `usage_table` *entries added in exercises* S354e⟩≡ (S322b)

```
{ ANEXP(BREAKX),     "(break)" },
{ ANEXP(CONTINUEX),  "(continue)" },
{ ANEXP(RETURNX),    "(return exp)" },
{ ANEXP(THROW),      "(throw lbl-name exp)" },
{ ANEXP(TRY_CATCH),  "(try-catch body lbl-name handler)" },
{ ANEXP(LONG_LABEL), "(long-label lbl-name body)" },
{ ANEXP(LONG_GOTO),  "(long-goto lbl-name exp)" },
```

The `smartBegin` function knows that `(begin` e`)` is always and forever equivalent to e, and it prefers the simpler version.

S354f. ⟨*lowering functions for μScheme+* S354f⟩≡ (S323a)

```
static Exp smartBegin(Explist es) {
    if (es != NULL && es->tl == NULL)
        return es->hd;
    else
        return mkBegin(es);
}
```

S354g. ⟨`reduce_to_xdef` *case for* `ADEF(DEFINE)` **⟦uschemeplus⟧** S354g⟩≡

```
case ADEF(DEFINE):
  return mkDef(mkDefine(out[0].name,
                        mkLambda(out[1].names, smartBegin(out[2].exps))));
```

S354h. ⟨*extend-syntax.c* S354h⟩≡

```
extern void extendDefine(void);
void extendSyntax(void) { extendDefine(); }
```

S355a. ⟨*extra cases for printing μScheme ASTs* S355a⟩≡ (S339a) S355b ▷

```
  case BREAKX:
      bprint(output, "(break)");
      break;
  case CONTINUEX:
      bprint(output, "(continue)");
      break;
  case RETURNX:
      bprint(output, "(return %e)", e->returnx);
      break;
  case THROW:
      bprint(output, "(throw %n %e)", e->throw.label, e->throw.exp);
      break;
  case TRY_CATCH:
      bprint(output, "(try-catch %e %n %e)", e->try_catch.body,
                     e->try_catch.label, e->try_catch.handler);
      break;
  case LONG_LABEL:
      bprint(output, "(long-label %n %e)", e->long_label.label, e->long_label.body
      break;
  case LONG_GOTO:
      bprint(output, "(long-goto %n %e)", e->long_goto.label, e->long_goto.exp);
      break;
  case HOLE:
      bprint(output, "<*>");
      break;
  case ENV:
      bprint(output, "Saved %senvironment %*",
             e->env.tag == CALL ? "caller's " : "", (void*)e->env.contents);
      break;
```

S355b. ⟨*extra cases for printing μScheme ASTs* S355a⟩+≡ (S339a) ◁ S355a

```
  case LOWERED:
      bprint(output, "%e", e->lowered.before);
      break;
  case LOOPBACK:
      bprint(output, "...loopback...");
      break;
```

M.5 Finding free variables

In case the interpreter needs to print the free variables of closures, each new syntactic form needs a case in the `freevars` function from Appendix L.

S355c. ⟨*extra cases for finding free variables in μScheme expressions* S355c⟩≡ (S329a) S356a ▷

```
  case BREAKX:
      break;
  case CONTINUEX:
      break;
  case RETURNX:
      free = freevars(e->returnx, bound, free);
      break;
  case THROW:
      free = freevars(e->throw.exp, bound, free);
      break;
  case TRY_CATCH:
      free = freevars(e->try_catch.body, bound, free);
      free = freevars(e->try_catch.handler, bound, free);
      break;
```

S356a. ⟨*extra cases for finding free variables in µScheme expressions* S355c⟩+≡ (S329a) ◁ S355c S356b ▷

```
case LONG_LABEL:
    free = freevars(e->long_label.body, bound, free);
    break;
case LONG_GOTO:
    free = freevars(e->long_goto.exp, bound, free);
    break;
case LOWERED:
    free = freevars(e->lowered.before, bound, free);
             // dare not look at after, because it might loop
    break;
case LOOPBACK:
    break;
```

The remaining new forms appear only in contexts. A context should never appear in a closure, so if the interpreter finds a hole or an environment, it halts with an assertion failure.

S356b. ⟨*extra cases for finding free variables in µScheme expressions* S355c⟩+≡ (S329a) ◁ S356a

```
case HOLE:
case ENV:
    assert(0);
    break;
```

M.6 Options and other leftover bits

This section shows bits of interpreter code that don't fit anywhere else.

The µScheme+ interpreter has two internal options that can be changed during execution: whether to optimize tail calls and whether to show the high stack mark after each evaluation. These options are controlled by µScheme variables, the values of which are obtained from the global environment by function `getoption`.

S356c. ⟨*options.c* S356c⟩≡

```
Value getoption(Name name, Env env, Value defaultval) {
    Value *p = find(name, env);
    if (p)
        return *p;
    else
        return defaultval;
}
```

Stack-tracking options are set above. The remaining option is the one that controls the optimization of tail calls. By default, tail calls are optimized.

S356d. ⟨*use the options in* `env` *to initialize the instrumentation* S346e⟩+≡ (227a) ◁ S347c

```
optimize_tail_calls =
    istrue(getoption(strtoname("&optimize-tail-calls"), env, truev));
```

Interpreters defined in Chapter 4 check value pointers for validity. But in Chapter 3, no validation is needed.

S356e. ⟨*validate.c* S356e⟩≡

```
Value validate(Value v) {
    return v;
}
```

When the interpreter is evaluating a current expression, that expression should never be a hole or an environment. If it is, the interpreter halts with an assertion failure.

S357a. ⟨*expression-evaluation cases for forms that appear only as frames* S357a⟩≡ (228a)

```
case HOLE:
case ENV:
    assert(0);
```

And when the interpreter is evaluating a current value, the top of the stack should be a well-formed context. If the top of the stack takes any of the forms below, then it does not represent a well-formed context, and the interpreter halts with an assertion failure.

S357b. ⟨*cases for forms that never appear as frames* S357b⟩≡ (228b)

```
case LITERAL:  // syntactic values never appear as frames
case VAR:
case LAMBDAX:
case HOLE:     // and neither do bare holes
case BREAKX:   // nor does sugar
case CONTINUEX:
case WHILEX:
case BEGIN:
case TRY_CATCH:
case THROW:
case LOWERED:
case LOOPBACK:
    assert(0);
```

The interpreter uses a lot of holes, and I don't want it to have to allocate each one. Instead, I define a single static value `hole`, which is stored in initialized data, not on the heap.

S357c. ⟨*definition of static* `Exp hole`*, which always has a hole* S357c⟩≡ (227a)

```
static struct Exp holeExp = { HOLE, { { NIL, { 0 } } } };
static Exp hole = &holeExp;
```

The implementation of `letrec` begins by allocating a new mutable cell for each bound variable. The cell is initialized to an unspecified value.

S357d. ⟨*bind every name in* `e->letx.xs` *to an unspecified value in* `env` S357d⟩≡ (235a)

```
{   Namelist xs;
    for (xs = e->letx.xs; xs; xs = xs->tl)
        env = bindalloc(xs->hd, unspecified(), env);
}
```

All the interpreter code has now been presented, except for the code that gathers all the interfaces into the `all.h` header file.

S358. ⟨`all.h` *for μScheme+* S358⟩≡

```
#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <inttypes.h>
#include <limits.h>
#include <setjmp.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef __GNUC__
#define __noreturn __attribute__((noreturn))
#else
#define __noreturn
#endif

⟨early type definitions for μScheme S309c⟩
⟨type definitions for μScheme+ 223⟩
⟨type definitions for μScheme 145b⟩
⟨shared type definitions 43a⟩

⟨structure definitions for μScheme+ 224a⟩
⟨structure definitions for μScheme S322a⟩
⟨shared structure definitions (from chunk 697b)⟩

⟨function prototypes for μScheme+ S345d⟩
⟨function prototypes for μScheme 153b⟩
⟨shared function prototypes 43b⟩

⟨global variables for μScheme+ 224e⟩

⟨macro definitions used in parsing (from chunk 697b)⟩
⟨declarations of globals used in lexical analysis and parsing (from chunk 697b)⟩
```

M.7 Deleted scene: Delimited continuations

The last part of Chapter 3 mentions *continuations* (page 242). The `call/cc` primitive described there implements *undelimited* continuations. An undelimited continuation captures the entire evaluation stack—everything that is expected to happen in the future execution of the program. But in many situations, it is more interesting to capture only *part* of the stack, up to a *delimiter*: a *delimited continuation*.

The delimited-continuation primitives that best fit the semantics of Chapter 3 are called `prompt` and `control`.

- A `prompt` marks a spot on the stack. It's a bit like a `catch` with no handler.
- Like `call/cc`, `control` captures the current evaluation context—but only up to the nearest `prompt`. The `prompt` acts as a *delimiter* which limits the extent of the continuation that is captured.

Crucially, "capturing" a continuation does not mean copying it. The stack is not copied; instead, the part of the stack between the `control` and the `prompt` is *moved* into a continuation value.

- Equally crucially, when a continuation is called as a function, its stack does *not replace* the current context. Instead, the saved stack is *pushed on top of* the current context.

The `prompt` and `control` primitives honor the correspondence between evaluation contexts and functions: unlike the undelimited continuations captured by `call/cc`, the delimited continuations captured with `control` compose nicely with themselves and with ordinary functions. The primitives are described by these rules:

$$\frac{}{\langle \textsc{prompt}(e), \rho, \sigma, S\rangle \rightarrow \langle e, \rho, \sigma, \textsc{prompt}(\bullet) :: S\rangle} \qquad (\textsc{Prompt})$$

$$\frac{}{\langle v, \rho, \sigma, \textsc{prompt}(\bullet) :: S\rangle \rightarrow \langle v, \rho, \sigma, S\rangle} \qquad (\textsc{Prompt-Finish})$$

$$\frac{}{\langle \textsc{control}(e), \rho, \sigma, S\rangle \rightarrow \langle e, \rho, \sigma, \textsc{control}(\bullet) :: S\rangle} \qquad (\textsc{Control})$$

$$\frac{v_f \text{ is a function} \qquad \text{None of } F_1, \ldots, F_n \text{ has the form } \textsc{prompt}(\bullet)}{\begin{array}{l}\langle v_f, \rho, \sigma, \textsc{control}(\bullet) :: F_1 :: \cdots :: F_n :: \textsc{prompt}(\bullet) :: S\rangle \rightarrow \\ \quad \langle \textsc{apply}(v_f, \textsc{continuation}(F_1, \ldots, F_n)), \rho, \sigma, \textsc{prompt}(\bullet) :: S\rangle\end{array}}$$
(Control-Capture)

$$\frac{v_f = \textsc{continuation}(F_1, \ldots, F_n)}{\langle v_1, \rho, \sigma, \textsc{apply}(v_f, \bullet) :: S\rangle \rightarrow \langle v_1, \rho, \sigma, F_1 :: \cdots :: F_n :: S\rangle}$$
(Apply-Delimited-Continuation)

M.8 Bonus exercises

This section presents two exercises that are worthy but that didn't make the final cut for Chapter 3.

26. I claim that μScheme$+$ is a *conservative extension* of μScheme. This means that every μScheme definition is a value μScheme$+$ definition, and that every such definition has the same effect in μScheme$+$ as it has in μScheme. (Because an expression is also a definition, the same holds of expressions.)

    This claim can be made formal and can be backed up with proof. The first part of the claim is as follows:

    > Whenever the μScheme rules can prove $\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$, there is a ρ' such that $\langle e, \rho, \sigma, []\rangle \rightarrow^* \langle v, \rho', \sigma', []\rangle$.

    A proof of this claim needs an induction hypothesis, which is slightly stronger than the claim:

    > Whenever the μScheme rules can prove $\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$, there exists a ρ' such that for every stack S, $\langle e, \rho, \sigma, S\rangle \rightarrow^* \langle v, \rho', \sigma', S\rangle$.

    The claim is proved by induction over the derivation of $\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$.

    (a) Prove base cases for LITERAL, VAR, and LAMBDA.

    (b) Prove the induction step for a derivation that ends in an instance of Big-Step-Assign.

(c) Prove the induction steps for derivations that end in an instance of BIG-STEP-IFTRUE or BIG-STEP-IFFALSE.

(d) Prove the induction step for a derivation that ends in an instance of BIG-STEP-APPLYCLOSURE, for the special case that there is exactly one argument expression e_1 in the APPLY node.

(e) Prove the induction step for a derivation that ends in an instance of BIG-STEP-WHILEEND.

(f) Prove the induction step for a derivation that ends in an instance of BIG-STEP-WHILEITERATE.

So far the only claim I've made formal is that if an expression e can be evaluated in μScheme, then in μScheme$+$, e is evaluated in the same way. For μScheme$+$ to be considered a true conservative extension, we also have to be sure it doesn't add any behaviors:

> If given e, ρ, and σ, there do not exist a v and σ' such that $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$, then there does not exist a ρ' and σ' such that $\langle e, \rho, \sigma, [] \rangle \rightarrow^* \langle v, \rho', \sigma', [] \rangle$.

The techniques needed to prove this half of the claim are beyond the scope of this book.

27. In an evaluation context, a sequence of the form $v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n$ is represented as a value of type `Explist`. When it's time to transition to the next context, finding the hole takes time proportional to i. That means the total work involved in evaluating the sequence is about $\frac{1}{2}n^2$. In most programs, n is so small that this doesn't matter. But for the sake of craftsmanship, change the representation of these contexts to be a pair of lists $v_{i-1}, v_{i-1}, \ldots, v_1$ and $e_{i+1}, \ldots, e_n$.[1] Expect these changes:

- Transition from one context to another takes constant time and space.
- No `Explist` is ever copied. Memory management gets simpler, and the system allocates less memory overall.
- When the context is complete, the list of values needed is in reverse order. To cut down on further memory allocation, consider reversing the list by mutating pointers in place.

When you're done, answer these questions:

(a) Given a long-running μScheme program, can you measure any reproducible difference in the performance of the two interpreters?

(b) If you can use a memory-analysis tool like Valgrind, what changes do you measure in the amount of allocation? What about the amount of memory "lost" at the end of execution?

(c) If you were building a new system from scratch, which method would you use? Why?

[1] To save yourself the massive headache of changing the representations of all the contexts, define C macros or `static inline` functions to convert between an `Explist` pointer and a pointer to your pair of lists.

Appendix N contents

Supporting code for garbage collection N

This appendix shows supporting code that can help with the exercises in Chapter 4. The code includes functions that visit the data structures used in the mark-and-sweep collector, functions that scan the data structures used in the copying collector, code that helps the evaluator keep track of roots, and code that implements the root stack.

N.1 Basic support for the two collectors

N.1.1 *Object-visiting procedures for mark-and-sweep collection*

During mark-and-sweep collection, μScheme objects are visited in a depth-first search. Each type of object is visited by its own procedure. A few such procedures are presented in Chapter 4 (pages 268 to 270). The rest are here.

Function `visitexp` visits all literal values reachable from an expression `e`. That means visiting `e`'s literal value, if `e` is a literal, and of course `e`'s subexpressions, if any.

S363a. ⟨*ms.c* S363a⟩≡ S364d ▷

```
static void visitexp(Exp e) {
    switch (e->alt) {
    ⟨cases for visitexp S363b⟩
    }
    assert(0);
}
```

There are more cases than will fit on a page, so I break them into groups. First, some μScheme expressions:

S363b. ⟨*cases for* `visitexp` S363b⟩≡ (S363a) S364a ▷

```
case LITERAL:
    visitvalue(e->literal);
    return;
case VAR:
    return;
case IFX:
    visitexp(e->ifx.cond);
    visitexp(e->ifx.truex);
    visitexp(e->ifx.falsex);
    return;
case WHILEX:
    visitexp(e->whilex.cond);
    visitexp(e->whilex.body);
    return;
case BEGIN:
    visitexplist(e->begin);
    return;
```

The remaining μScheme expressions.

S364a. ⟨*cases for* `visitexp` S363b⟩+≡ (S363a) ◁ S363b S364b ▷

```
case SET:
    visitexp(e->set.exp);
    return;
case LETX:
    visitexplist(e->letx.es);
    visitexp(e->letx.body);
    return;
case LAMBDAX:
    visitexp(e->lambdax.body);
    return;
case APPLY:
    visitexp(e->apply.fn);
    visitexplist(e->apply.actuals);
    return;
```

Next, the μScheme+ expressions:

S364b. ⟨*cases for* `visitexp` S363b⟩+≡ (S363a) ◁ S364a S364c ▷

```
case BREAKX:
    return;
case CONTINUEX:
    return;
case RETURNX:
    visitexp(e->returnx);
    return;
case THROW:
    visitexp(e->throw.exp);
    return;
case TRY_CATCH:
    visitexp(e->try_catch.handler);
    visitexp(e->try_catch.body);
    return;
case LONG_GOTO:
    visitexp(e->long_goto.exp);
    return;
case LONG_LABEL:
    visitexp(e->long_label.body);
    return;
case LOWERED:
    visitexp(e->lowered.before);
    return;
case LOOPBACK:
    return;
```

Last, μScheme+ evaluation contexts:

S364c. ⟨*cases for* `visitexp` S363b⟩+≡ (S363a) ◁ S364b

```
case ENV:
    visitenv(e->env.contents);
    return;
case HOLE:
    return;
```

A list of expressions is visited by `visitexplist`.

S364d. ⟨*ms.c* S363a⟩+≡ ◁ S363a S365a ▷

```
static void visitexplist(Explist es) {
    for (; es; es = es->tl)
        visitexp(es->hd);
}
```

A list of registers is visited by `visitregisterlist`.

S365a. ⟨*ms.c* S363a⟩+≡ ◁S364d S365b▷

```
static void visitregisterlist(Registerlist regs) {
    for ( ; regs != NULL; regs = regs->tl)
        visitregister(regs->hd);
}
```

A `Stack` is visited by visiting every frame. A frame can be seen only if the representation is exposed, so ⟨*representation of* `struct Stack` S343a⟩ is defined here.

S365b. ⟨*ms.c* S363a⟩+≡ ◁S365a S365c▷

```
⟨representation of struct Stack S343a⟩
static void visitstack(Stack s) {
    Frame *fr;
    for (fr = s->frames; fr < s->sp; fr++) {
        visitframe(fr);
    }
}
```

Visiting a frame means visiting both of its expressions.

S365c. ⟨*ms.c* S363a⟩+≡ ◁S365b S365d▷

```
static void visitframe(Frame *fr) {
    visitexp(&fr->form);
    if (fr->syntax != NULL)
        visitexp(fr->syntax);
}
```

Visiting lists of pending unit tests means visiting all tests on the list.

S365d. ⟨*ms.c* S363a⟩+≡ ◁S365c S365e▷

```
static void visittestlists(UnitTestlistlist uss) {
    UnitTestlist ul;

    for ( ; uss != NULL; uss = uss->tl)
        for (ul = uss->hd; ul; ul = ul->tl)
            visittest(ul->hd);
}
```

Visiting a unit test means visiting its component expressions.

S365e. ⟨*ms.c* S363a⟩+≡ ◁S365d S366a▷

```
static void visittest(UnitTest t) {
    switch (t->alt) {
    case CHECK_EXPECT:
        visitexp(t->check_expect.check);
        visitexp(t->check_expect.expect);
        return;
    case CHECK_ASSERT:
        visitexp(t->check_assert);
        return;
    case CHECK_ERROR:
        visitexp(t->check_error);
        return;
    }
    assert(0);
}
```

Visiting roots means visiting the global variables, the stack, and any machine registers.

S366a. ⟨*ms.c* S363a⟩+≡ ◁S365e

```
static void visitroots(void) {
    visitenv(*roots.globals.user);
    visittestlists(roots.globals.internal.pending_tests);
    visitstack(roots.stack);
    visitregisterlist(roots.registers);
}
```

N.1.2 Root-scanning procedures for copying collection

Copying collection begins by scanning roots and forwarding them to to-space. In Chapter 4 (chunk 273), the scanning actions are written using Noweb chunks. Each chunk calls one scanning procedure.

S366b. ⟨*scan frame* `*fr`*, forwarding all internal pointers* S366b⟩≡ (273)

```
scanframe(fr);
```

S366c. ⟨*scan list of unit tests* `testss->hd`*, forwarding all internal pointers* S366c⟩≡ (273)

```
scantests(testss->hd);
```

S366d. ⟨*scan register* `regs->hd`*, forwarding all internal pointers* S366d⟩≡ (273)

```
scanloc(regs->hd);
```

S366e. ⟨*scan object* `*scanp`*, forwarding all internal pointers* S366e⟩≡ (273)

```
scanloc(scanp);
```

Each type of object has its own scanning procedure. A few such procedures are presented in Chapter 4 (pages 276 to 277), and the rest appear below. As explained in Chapter 4, these scanning procedures are hybrids. Like standard scanning procedures, they forward internal pointers to objects allocated on the μScheme heap. But because some potential roots are allocated on the C heap, these procedures use graph traversal to visit those. Almost all the forwarding is done by `scanloc`, which is shown in Chapter 4 (chunk 277c). The remaining procedures that are shown here either call `scanloc`, do graph traversal, or both. These procedures are therefore very similar to the visiting procedures in the previous section.

Expressions are scanned by scanning internal values or subexpressions.

S366f. ⟨*copy.c* S366f⟩≡ S368b▷

```
static void scanexp(Exp e) {
    switch (e->alt) {
    ⟨cases for scanexp S366g⟩
    }
    assert(0);
}
```

First, μScheme expressions:

S366g. ⟨*cases for* `scanexp` S366g⟩≡ (S366f) S367a▷

```
case LITERAL:
    scanloc(&e->literal);
    return;
case VAR:
    return;
case IFX:
    scanexp(e->ifx.cond);
    scanexp(e->ifx.truex);
    scanexp(e->ifx.falsex);
    return;
```

More μScheme expressions.

S367a. ⟨*cases for* scanexp S366g⟩+≡ (S366f) ◁ S366g S367b ▷

```
case WHILEX:
    scanexp(e->whilex.cond);
    scanexp(e->whilex.body);
    return;
case BEGIN:
    scanexplist(e->begin);
    return;
case SET:
    scanexp(e->set.exp);
    return;
case LETX:
    scanexplist(e->letx.es);
    scanexp(e->letx.body);
    return;
case LAMBDAX:
    scanexp(e->lambdax.body);
    return;
case APPLY:
    scanexp(e->apply.fn);
    scanexplist(e->apply.actuals);
    return;
```

Next, μScheme+ expressions:

S367b. ⟨*cases for* scanexp S366g⟩+≡ (S366f) ◁ S367a S368a ▷

```
case BREAKX:
    return;
case CONTINUEX:
    return;
case RETURNX:
    scanexp(e->returnx);
    return;
case THROW:
    scanexp(e->throw.exp);
    return;
case TRY_CATCH:
    scanexp(e->try_catch.handler);
    scanexp(e->try_catch.body);
    return;
case LONG_GOTO:
    scanexp(e->long_goto.exp);
    return;
case LONG_LABEL:
    scanexp(e->long_label.body);
    return;
case LOWERED:
    scanexp(e->lowered.before);
    scanexp(e->lowered.after);
    return;
case LOOPBACK:
    return;
```

type Exp	$\mathcal{A}$
roots	265b
scanexp	277a
scanexplist	277a
scanloc	277a
visitenv	268c
visitregisterlist	268c
visitstack	268c
visittestlists	268c

Last, *μ*Scheme+ evaluation contexts.

S368a. ⟨*cases for* scanexp S366g⟩+≡ (S366f) ◁S367b

```
case HOLE:
    return;
case ENV:
    scanenv(e->env.contents);
    return;
```

A frame is scanned by scanning its expressions.

S368b. ⟨*copy.c* S366f⟩+≡ ◁S366f S368c▷

```
static void scanframe(Frame *fr) {
    scanexp(&fr->form);
    if (fr->syntax != NULL)
        scanexp(fr->syntax);
}
```

A list of expressions is scanned by scanexplist.

S368c. ⟨*copy.c* S366f⟩+≡ ◁S368b S368d▷

```
static void scanexplist(Explist es) {
    for (; es; es = es->tl)
        scanexp(es->hd);
}
```

A source is scanned by scanning its pending tests.

S368d. ⟨*copy.c* S366f⟩+≡ ◁S368c S368e▷

```
static void scantests(UnitTestlist tests) {
    for (; tests; tests = tests->tl)
        scantest(tests->hd);
}
```

A test is scanned by scanning its expressions.

S368e. ⟨*copy.c* S366f⟩+≡ ◁S368d

```
static void scantest(UnitTest t) {
    switch (t->alt) {
    case CHECK_EXPECT:
        scanexp(t->check_expect.check);
        scanexp(t->check_expect.expect);
        return;
    case CHECK_ASSERT:
        scanexp(t->check_assert);
        return;
    case CHECK_ERROR:
        scanexp(t->check_error);
        return;
    }
    assert(0);
}
```

N.1.3 Access to the desired size of the heap

The size of the garbage-collected heap can be controlled by setting the μScheme variable `&gamma-desired`, as described in Exercises 3 and 10 in Chapter 4. The value of that variable, if any, is fetched by function `gammadesired`.

S369a. ⟨*loc.c* S369a⟩≡ S372e ▷

```
int gammadesired(int defaultval, int minimum) {
    assert(roots.globals.user != NULL);
    Value *gammaloc = find(strtoname("&gamma-desired"), *roots.globals.user);
    if (gammaloc && gammaloc->alt == NUM)
        return gammaloc->num > minimum ? gammaloc->num : minimum;
    else
        return defaultval;
}
```

S369b. ⟨*function prototypes for μScheme* S369b⟩≡ (S318a S358)

```
int gammadesired(int defaultval, int minimum);
```

N.1.4 Code to push and pop register roots

The global `roots` data structure holds global variables (including pending unit tests), a stack and a list of "machine registers." All are initially `NULL`.

S369c. ⟨*root.c* S369c⟩≡ S369d ▷

```
struct Roots roots = { { NULL, { NULL } }, NULL, NULL };
```

Register roots are pushed and popped by `pushreg` and `popreg`.

S369d. ⟨*root.c* S369c⟩+≡ ◁ S369c S369e ▷

```
void pushreg(Value *reg) {
    roots.registers = mkRL(reg, roots.registers);
}
```

When a register is popped, `popreg` insists that the register being popped actually be present at the head of the register list.

S369e. ⟨*root.c* S369c⟩+≡ ◁ S369d S369f ▷

```
void popreg(Value *reg) {
    Registerlist regs = roots.registers;
    assert(regs != NULL);
    assert(reg == regs->hd);
    roots.registers = regs->tl;
    free(regs);
}
```

When a list of registers is pushed, that list must be popped in the opposite order (last in, first out). Here registers are pushed left to right and popped right to left.

S369f. ⟨*root.c* S369c⟩+≡ ◁ S369e

```
void pushregs(Valuelist regs) {
    for (; regs; regs = regs->tl)
        pushreg(&regs->hd);
}

void popregs (Valuelist regs) {
    if (regs != NULL) {
        popregs(regs->tl);
        popreg(&regs->hd);
    }
}
```

type `Explist`	S309b
`find`	153b
type `Frame`	223
`mkRL`	$\mathcal{A}$
`popreg`	265c
`popregs`	265d
`pushreg`	265c
type `Registerlist`	S377b
`roots`	265b
`scanenv`	277a
`scanexp`	277a
`scantest`	277a
`strtoname`	43b
type `UnitTest`	$\mathcal{A}$
type `UnitTestlist`	S309b
type `Value`	$\mathcal{A}$
type `Valuelist`	S309c

N.2 GC debugging, with or without Valgrind

As I write, the gold standard for debugging memory errors is Valgrind (or a similar tool like Pin). Valgrind's Memcheck tool tracks the status of every bit of every word of memory; it knows what bits have been initialized and what bits haven't been initialized, and it knows which parts of memory the program has permission to read and write. Although Memcheck was designed to find general memory errors in unsafe programs, not specifically to debug garbage collectors, it is still an invaluable asset. And with the help of the code below, Valgrind can be used implement the debugging interface described in Section 4.6.1. (If you don't have Valgrind, you can turn it off with one C macro.)

This code implements the debugging interface described in Section 4.6.1. It finds bugs in three ways:

- When memory belongs to the collector and not the interpreter, the `alt` field is set to `INVALID`. If `validate` is called with an `INVALID` expression, it dies.
- When memory belongs to the collector and not the interpreter, Valgrind is informed that nobody must read or write it. If your collector mistakenly reclaims memory that the interpreter still has access to, when the interpreter tries to read or write that memory, Valgrind will bleat. (Valgrind is discussed briefly in Section 4.10.2, page 291.)
- When memory is given from the collector to the interpreter, Valgrind is informed that it is OK to write but not OK to read until it has been initialized.

If you don't have Valgrind, you can `#define NOVALGRIND`, and you'll still have the `INVALID` thing in the `alt` field to help you.

S370a. ⟨*gcdebug.c* S370a⟩≡ S371a ▷

```
#ifndef NOVALGRIND
  #include <valgrind/memcheck.h>
#else
  ⟨define do-nothing replacements for Valgrind macros S370b⟩
#endif
```

To prevent compiler warnings, the do-nothing macros "evaluate" their arguments by casting them to `void`.

S370b. ⟨*define do-nothing replacements for Valgrind macros* S370b⟩≡ (S370a)

```
#define VALGRIND_CREATE_BLOCK(p, n, s)     ((void)(p),(void)(n),(void)(s))
#define VALGRIND_CREATE_MEMPOOL(p, n, z)   ((void)(p),(void)(n),(void)(z))
#define VALGRIND_MAKE_MEM_DEFINED_IF_ADDRESSABLE(p, n) \
                                           ((void)(p),(void)(n))
#define VALGRIND_MAKE_MEM_DEFINED(p, n)    ((void)(p),(void)(n))
#define VALGRIND_MAKE_MEM_UNDEFINED(p, n)  ((void)(p),(void)(n))
#define VALGRIND_MAKE_MEM_NOACCESS(p, n)   ((void)(p),(void)(n))
#define VALGRIND_MEMPOOL_ALLOC(p1, p2, n)  ((void)(p1),(void)(p2),(void)(n))
#define VALGRIND_MEMPOOL_FREE(p1, p2)      ((void)(p1),(void)(p2))
```

Valgrind gets its information via calls that are described in Valgrind's documentation for "custom memory allocators."

When the heap is initialized, my code creates a `gc_pool`, which stands for all objects allocated using `allocloc`. The flag `gc_uses_mark_bits`, if set, tells Valgrind that when memory is first allocated, its contents are zero. My code also initializes the `gcverbose` flag.

S370c. ⟨*global variables used in garbage collection* S370c⟩≡ (S377d)

```
extern bool gc_uses_mark_bits;
```

S371a. ⟨*gcdebug.c* S370a⟩+≡ ◁S370a S371b▷

```
static int gc_pool_object;
static void *gc_pool = &gc_pool_object;  /* valgrind needs this */
static int gcverbose;  /* GCVERBOSE tells gcprintf & gcprint to make noise */

void gc_debug_init(void) {
    VALGRIND_CREATE_MEMPOOL(gc_pool, 0, gc_uses_mark_bits);
    gcverbose = getenv("GCVERBOSE") != NULL;
}
```

When new objects are acquired from the operating system, each one is marked invalid and is made known to Valgrind. Because the objects' memory belongs to the collector, it is marked as inaccessible.

S371b. ⟨*gcdebug.c* S370a⟩+≡ ◁S371a S371c▷

```
void gc_debug_post_acquire(Value *mem, unsigned nvalues) {
    unsigned i;
    for (i = 0; i < nvalues; i++) {
        gcprintf("ACQUIRE %p\n", (void*)&mem[i]);
        mem[i] = mkInvalid("memory acquired from OS");
        VALGRIND_CREATE_BLOCK(&mem[i], sizeof(*mem), "managed Value");
    }
    ⟨when using mark bits, barf unless nvalues is 1 S372c⟩
    VALGRIND_MAKE_MEM_NOACCESS(mem, nvalues * sizeof(*mem));
}
```

Before memory is released, the code below checks that each object is invalid. Valgrind has to be told that it's temporarily OK to look at the object.

S371c. ⟨*gcdebug.c* S370a⟩+≡ ◁S371b S371d▷

```
void gc_debug_pre_release(Value *mem, unsigned nvalues) {
    unsigned i;
    for (i = 0; i < nvalues; i++) {
        gcprintf("RELEASE %p\n", (void*)&mem[i]);
        VALGRIND_MAKE_MEM_DEFINED(&mem[i].alt, sizeof(mem[i].alt));
        assert(mem[i].alt == INVALID);
    }
    VALGRIND_MAKE_MEM_NOACCESS(mem, nvalues * sizeof(*mem));
}
```

Before an object is handed to the interpreter, Valgrind is told that the object has been allocated, then the object is made invalid, and finally Valgrind is told that the object is writable but uninitialized.

S371d. ⟨*gcdebug.c* S370a⟩+≡ ◁S371c S372a▷

```
void gc_debug_pre_allocate(Value *mem) {
    gcprintf("ALLOC %p\n", (void*)mem);
    VALGRIND_MEMPOOL_ALLOC(gc_pool, mem, sizeof(*mem));
    VALGRIND_MAKE_MEM_DEFINED_IF_ADDRESSABLE(&mem->alt, sizeof(mem->alt));
    assert(mem->alt == INVALID);
    *mem = mkInvalid("allocated but uninitialized");
    VALGRIND_MAKE_MEM_UNDEFINED(mem, sizeof(*mem));
}
```

gc_uses_mark_bits	S377f
gcprintf	282b
mkInvalid	𝒜
type Value	𝒜

When an object is reclaimed, it should *not* be invalid—because it should have been initialized to a valid value immediately after it was allocated. After the object's validity is confirmed, the object is marked invalid, and Valgrind is told that the object has been freed.

S372a. ⟨*gcdebug.c* S370a⟩+≡ ◁S371d S372b▷

```
void gc_debug_post_reclaim(Value *mem) {
    gcprintf("FREE %p\n", (void*)mem);
    assert(mem->alt != INVALID);
    *mem = mkInvalid("memory reclaimed by the collector");
    VALGRIND_MEMPOOL_FREE(gc_pool, mem);
}
```

Objects in a block can be reclaimed in one call, *provided* the block is an array of `Value`, not an array of `Mvalue`.

S372b. ⟨*gcdebug.c* S370a⟩+≡ ◁S372a S373a▷

```
void gc_debug_post_reclaim_block(Value *mem, unsigned nvalues) {
    unsigned i;
    ⟨when using mark bits, barf unless nvalues is 1 S372c⟩
    for (i = 0; i < nvalues; i++)
        gc_debug_post_reclaim(&mem[i]);
}
```

S372c. ⟨*when using mark bits, barf unless* `nvalues` *is 1* S372c⟩≡ (S371b 372b)

```
if (gc_uses_mark_bits) /* mark and sweep */
    assert(nvalues == 1);
```

Function `validate` is used freely in the interpreter to make sure all values are good. Calling `validate(v)` returns `v`, unless `v` is invalid, in which case it causes an assertion failure.

S372d. ⟨*validate.c* S372d⟩≡

```
Value validate(Value v) {
    assert(v.alt != INVALID);
    return v;
}
```

When the collector is initialized, function `initallocate` uses the ANSI C function `atexit` to make sure that before the program exits, final garbage-collection statistics are printed.

S372e. ⟨*loc.c* S369a⟩+≡ ◁S369a

```
extern void printfinalstats(void);
void initallocate(Env *globals) {
    gc_debug_init();
    roots.globals.user                   = globals;
    roots.globals.internal.pending_tests = NULL;
    roots.stack     = emptystack();
    roots.registers = NULL;
    atexit(printfinalstats);
}
```

Functions that print GC diagnostics are defined here.

S373a. ⟨*gcdebug.c* S370a⟩+≡ ◁ S372b S373b ▷

```
void gcprint(const char *fmt, ...) {
  if (gcverbose) {
    va_list_box box;
    Printbuf buf = printbuf();

    assert(fmt);
    va_start(box.ap, fmt);
    vbprint(buf, fmt, &box);
    va_end(box.ap);
    fwritebuf(buf, stderr);
    fflush(stderr);
    freebuf(&buf);
  }
}
```

S373b. ⟨*gcdebug.c* S370a⟩+≡ ◁ S373a S376c ▷

```
void gcprintf(const char *fmt, ...) {
  if (gcverbose) {
    va_list args;

    assert(fmt);
    va_start(args, fmt);
    vfprintf(stderr, fmt, args);
    va_end(args);
    fflush(stderr);
  }
}
```

N.3 Code that is changed to support garbage collection

When the garbage collector is deployed, most parts of the μScheme+ interpreter are either replaced completely or are used without change. But a few parts are modified versions of the originals. The modifications help to keep track of the root set: any code that can allocate is modified to make sure that before `allocloc` is called, the root set is up to date.

To keep the root set up to date, my code abuses the stack of evaluation contexts. If it needs to save an `Exp` or an `Env`, for example, my code pushes an appropriate context. Because the context is popped immediately after the allocation, these abusive contexts are never seen by the evaluator, and therefore they don't interfere with it. (If my code needs to save a `Value`, it simply uses `pushreg` or `pushregs` as intended.)

Code that is modified or added to support garbage collection is shown in *`typewriter italics`*.

N.3.1 Revised environment-extension routines

To be sure that the current environment is always visible to the garbage collector, μScheme+ needs a new version of `bindalloc`. When `bindalloc` is called, its `env` argument contains bindings to heap-allocated locations. And because `env` is a local variable in `eval`, it doesn't appear on the stack of evaluation contexts. It gets put on the stack so that when `allocate` is called, the bindings in `env` are kept live.

S374a. ⟨*env.c* S374a⟩≡ S374b ▷

```
Env bindalloc(Name name, Value val, Env env) {
    Env newenv = malloc(sizeof(*newenv));
    assert(newenv != NULL);

    newenv->name = name;
    pushframe(mkEnvStruct(env, NONCALL), roots.stack);
    newenv->loc  = allocate(val);
    popframe(roots.stack);
    newenv->tl   = env;
    return newenv;
}
```

Please also observe that `val` is a parameter passed by value, so `bindalloc` has a fresh copy of it. Because `val` contains `Value*` pointers, you might think it needs to be on the root stack for the copying collector (so that the pointers can be updated if necessary). But by the time `allocate` is called, this copy of `val` is dead—only `allocate`'s private copy matters.

In `bindalloclist`, by contrast, when `bindalloc` is called with `vs->hd`, not everything is dead. The value `vs->hd` is dead, as is everything that precedes it on list `vs`. But values reachable from `vs->tl` are still live. To make them visible to the garbage collector, `bindalloclist` treats the entire list `vs` as "machine registers."

S374b. ⟨*env.c* S374a⟩+≡ ◁ S374a

```
Env bindalloclist(Namelist xs, Valuelist vs, Env env) {
    Valuelist oldvals = vs;
    pushregs(oldvals);
    for (; xs && vs; xs = xs->tl, vs = vs->tl)
        env = bindalloc(xs->hd, vs->hd, env);
    popregs(oldvals);
    return env;
}
```

N.3.2 Revisions to `eval`

Chapter 3's `eval` function needs just one change to support garbage collection: it needs to make the evaluation stack part of the root set:

S374c. ⟨*ensure that* `evalstack` *is initialized and empty* S374c⟩≡ (227a)

```
assert(topframe(roots.stack) == NULL);
roots.stack = evalstack;
```

N.3.3 Revised `evaldef`

When given a VAL or DEFINE binding to a variable that is not already in the environment, `evaldef` has to extend the environment *before* evaluating the right-hand side. That means the right-hand side needs to be made a root—so `evaldef` pushes it onto the context stack. And because the garbage collector might move objects, after allocating, evaldef overwrites the original right-hand side with the version from the top of the stack.

S375a. ⟨*evaluate* val *binding and return new environment* S375a⟩≡ (159e)

```
{
    pushframe(*d->val.exp, roots.stack);
    if (find(d->val.name, env) == NULL)
        env = bindalloc(d->val.name, unspecified(), env);
    *d->val.exp = topframe(roots.stack)->form;
    popframe(roots.stack);
    Value v = eval(d->val.exp, env);
    *find(d->val.name, env) = v;
    ⟨if echo calls for printing, print either v or the bound name S311e⟩
    return env;
}
```

N.3.4 The revised parser

In a definition like

```
(reverse '(1 2 3 4 5))
```

the cons cells for the list are allocated on the heap *by the parser*. Since any expression might be a quoted S-expression, any call to parseexp can allocate. Therefore, before making a call to parseexp or parselist, or sExp or sExps, the parser must make sure that any quoted S-expression is visible as a root. As before, the code abuses the stack of evaluation contexts: if reduce_to_exp sees a quoted S-expression, it puts the quoted S-expression on the stack.

And it's the Exp (the pointer) that needs to be on the stack, not just the struct Exp. Putting the pointer on the stack is necessary so that if a copying garbage collection occurs, any pointers in the Exp get forwarded. Since the stack contains only values, not pointers, the parser gets the Exp pointer onto the stack by placing it inside a struct Exp that holds a BEGIN expression.

S375b. ⟨*parse.c* S375b⟩≡

```
Exp reduce_to_exp(int code, struct Component *comps) {
    switch(code) {
    case ANEXP(SET):     return mkSet(comps[0].name, comps[1].exp);
    case ANEXP(IFX):     return mkIfx(comps[0].exp, comps[1].exp, comps[2].exp);
    case ANEXP(BEGIN):   return mkBegin(comps[0].exps);
    ⟨cases for reduce_to_exp that are sugared in µScheme+ (from chunk 697b)⟩
    case ANEXP(LAMBDAX): return mkLambdax(mkLambda(comps[0].names, comps[1].exp)
    case ANEXP(APPLY):   return mkApply(comps[0].exp, comps[1].exps);
    case ANEXP(LITERAL):
    { Exp e = mkLiteral(comps[0].value);
      pushframe(mkBeginStruct(mkEL(e, NULL)), roots.stack);
      return e;
    }
    ⟨cases for µScheme's reduce_to_exp added in exercises S324g⟩
    }
    assert(0);
}
```

Expression e can't come off the stack until parsing is complete. It is actually left there until eval is called, at which point it is safe to remove it using clearstack.

The other part of the parser that has to change is the part that interprets a list as an S-Expression, as in '(a b c). In chunk S326b there's no garbage collector in play, so it simply calls parsesx on the hd and tl and then calls cons on the result. With a garbage collector in play, this simple code won't work: the second call might

allocate	154a
bindalloc	153c
type Env	153a
env	159e
eval	155a
evalstack	227a
type Exp	$\mathcal{A}$
find	153b
mkApply	$\mathcal{A}$
mkBegin	$\mathcal{A}$
mkBeginStruct	$\mathcal{A}$
mkEL	$\mathcal{A}$
mkEnvStruct	$\mathcal{A}$
mkIfx	$\mathcal{A}$
mkLambda	$\mathcal{A}$
mkLambdax	$\mathcal{A}$
mkLiteral	$\mathcal{A}$
mkSet	$\mathcal{A}$
type Name	43a
type Namelist	43a
popframe	224b
popregs	265d
pushframe	224b
pushregs	265d
roots	265b
topframe	224c
unspecified	154d
type Value	$\mathcal{A}$
type Valuelist	S309c

trigger a garbage collection, so the result of the first call has to be a root. That result goes (temporarily) into a "machine register."

S376a. ⟨*return* p->list *interpreted as an S-expression* S376a⟩≡ (S325d)

```
if (p->list == NULL)
    return mkNil();
else {
    Value v = parsesx(p->list->hd, source);
    pushreg(&v);
    Value w = parsesx(mkList(p->list->tl), source);
    popreg(&v);
    Value pair = cons(v, w);
    cyclecheck(&pair);
    return pair;
}
```

N.3.5 Checking for cycles in cons

In the early stages of debugging my collector, I found it useful to look for cycles: after every cons, my code would check to see if the newly allocated cons cell participated in a cycle. In case you might also find it helpful to check for cycles, I've kept the code.

S376b. ⟨*function prototypes for μScheme+* S376b⟩≡ (S358)

```
void cyclecheck(Value *l);
```

The code uses depth-first search to make sure no value is ever its own ancestor.

S376c. ⟨*gcdebug.c* S370a⟩+≡ ◁ S373b S376d ▷

```
struct va { /* value ancestors */
    Value *l;
    struct va *parent;
};
```

S376d. ⟨*gcdebug.c* S370a⟩+≡ ◁ S376c S377a ▷

```
static void check(Value *l, struct va *ancestors) {
    struct va *c;
    for (c = ancestors; c; c = c->parent)
        if (l == c->l) {
            fprintf(stderr, "%p is involved in a cycle\n", (void *)l);
            if (c == ancestors) {
                fprintf(stderr, "%p -> %p\n", (void *)l, (void *)l);
            } else {
                fprintf(stderr, "%p -> %p\n", (void *)l, (void *)ancestors->l);
                while (ancestors->l != l) {
                    fprintf(stderr, "%p -> %p\n",
                            (void *)ancestors->l, (void *)ancestors->parent->l);
                    ancestors = ancestors->parent;
                }
            }
            runerror("cycle of cons cells");
        }
}
```

S377a. ⟨*gcdebug.c* S370a⟩+≡ ◁S376d

```
static void search(Value *v, struct va *ancestors) {
    if (v->alt == PAIR) {
        struct va na;  // new ancestors
        check(v->pair.car, ancestors);
        check(v->pair.cdr, ancestors);
        na.l = v;
        na.parent = ancestors;
        search(v->pair.car, &na);
        search(v->pair.cdr, &na);
    }
}

void cyclecheck(Value *l) {
    search(l, NULL);
}
```

N.4 Type, structure, and value definitions for GC roots

The definition of the `Roots` data structure relies on types `Register`, `Registerlist`, and `UnitTestlistlist`.

S377b. ⟨*type definitions for μScheme+* S377b⟩≡ (S358)

```
typedef struct Value *Register;  // pointer to a local variable or a parameter
                                 // of a C function that could allocate
typedef struct Registerlist *Registerlist;         // list of Register
typedef struct UnitTestlistlist *UnitTestlistlist; // list of UnitTestlist (list)
```

The root type and its variables are visible to all C code.

S377c. ⟨*structure definitions for μScheme+* S377c⟩≡ (S358)
⟨*structure definitions used in garbage collection* 265a⟩

S377d. ⟨*global variables for μScheme+* S377d⟩≡ (S358)
⟨*global variables used in garbage collection* S370c⟩

N.5 Placeholders for exercises

The rest of this section includes the placeholder code that you are meant to replace when you do the implementation exercises.

S377e. ⟨*private declarations for copying collection* S377e⟩≡

```
static void collect(void);
```

S377f. ⟨*copy.c* **[[prototype]]** S377f⟩≡

```
/* you need to redefine these functions */
static void collect(void) { (void)scanframe; (void)scantests; assert(0); }
void printfinalstats(void) { assert(0); }
/* you need to initialize this variable */
bool gc_uses_mark_bits;
```

S377g. ⟨*ms.c* **[[prototype]]** S377g⟩≡ S377h ▷

```
/* you need to redefine these functions */
void printfinalstats(void) {
  (void)nalloc; (void)ncollections; (void)nmarks;
  assert(0);
}
```

S377h. ⟨*ms.c* **[[prototype]]** S377g⟩+≡ ◁S377g

```
void avoid_unpleasant_compiler_warnings(void) {
    (void)visitroots;
}
```

Appendix O contents

Supporting code for the ML interpreter for μScheme O

This appendix describes language-specific code that that is not interesting enough to include in Chapter 5, but that is needed to implement the ML version of μScheme. Code in this appendix defines some primitive functions, builds the initial basis, runs unit tests, and does lexical analysis and parsing. There is also code that implements the "unspecified" values in μScheme's operational semantics.

For every bridge language that is implemented in ML, this Supplement includes a similar appendix.

O.1 Organizing code chunks into an interpreter

My interpreter for μScheme takes up close to 2,000 lines of ML code. In the book, that code is broken up into small chunks. But those chunks are written into a single program—and according to the semantics of ML, every type and function must be defined before it is used. Definitions come not only from this appendix and Chapter 5 but also from Appendices H and I. To get all the definitions in the right order, I use Noweb code chunks.[1] Each interpreter is built from a different set of chunks, but they are all organized along the same lines: shared infrastructure; abstract syntax and values, with utility functions; lexical analysis and parsing; evaluation (including unit testing and the read-eval-print loop); and initialization. (Interpreters for typed languages also have chunks devoted to types and to type checking or type inference.)

```
S379. ⟨mlscheme.sml S379⟩≡
  ⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization S213a⟩

  ⟨abstract syntax and values for μScheme S380a⟩
  ⟨utility functions on values (μScheme, Typed μScheme, nano-ML) S380c⟩

  ⟨lexical analysis and parsing for μScheme, providing filexdefs and stringsxdefs S383c⟩

  ⟨evaluation, testing, and the read-eval-print loop for μScheme S380b⟩

  ⟨implementations of μScheme primitives and definition of initialBasis S382a⟩
  ⟨function runStream, which evaluates input given initialBasis S240b⟩
  ⟨look at command-line arguments, then run S240c⟩
```

[1] This technique is evidence of the biggest mistake I made in creating this software: I used Noweb code chunks, but I should have used ML modules. I knew I didn't want to expose readers to ML modules starting in Chapter 5, and that was a good decision. (Covering modules at all was a late change to the book's design.) But I wish I had used modules secretly, behind the scenes. Had I done so, the software would have been so much better.

The ⟨*abstract syntax and values for μScheme* S380a⟩ is itself defined as a sequence of smaller chunks.

S380a. ⟨*abstract syntax and values for μScheme* S380a⟩≡ (S379)

```
valueString      : value -> string
expString        : exp   -> string
```

⟨*definitions of* `exp` *and* `value` *for μScheme* 306⟩
⟨*definition of* `def` *for μScheme* 307a⟩
⟨*definition of* `unit_test` *for untyped languages (shared)* S214a⟩
⟨*definition of* `xdef` *(shared)* S214b⟩
⟨*definition of* `valueString` *for μScheme, Typed μScheme, and nano-ML* 307b⟩
⟨*definition of* `expString` *for μScheme* S383b⟩

Likewise the support for reading, evaluating, testing, and printing.

S380b. ⟨*evaluation, testing, and the read-eval-print loop for μScheme* S380b⟩≡ (S379)
⟨*definitions of* `eval`, `evaldef`, `basis`, *and* `processDef` *for μScheme* S238c⟩
⟨*shared unit-testing utilities* S225a⟩
⟨*shared definition of* `withHandlers` S239a⟩
⟨*definition of* `testIsGood` *for μScheme* S382c⟩
⟨*shared definition of* `processTests` S226⟩
⟨*shared read-eval-print loop* S237⟩

Different interpreters need different utility functions, but they all need an implementation of equality that can be used in `check-expect`. And the μScheme interpreter also needs an implementation of primitive equality. Primitive equality permits only atoms to be considered equal.

S380c. ⟨*utility functions on values (μScheme, Typed μScheme, nano-ML)* S380c⟩≡ (S379) S380d ▷

```
equalatoms : value * value -> bool
```

```
fun equalatoms (NIL,      NIL    )  = true
  | equalatoms (NUM  n1,  NUM  n2)  = (n1 = n2)
  | equalatoms (SYM  v1,  SYM  v2)  = (v1 = v2)
  | equalatoms (BOOLV b1, BOOLV b2) = (b1 = b2)
  | equalatoms  _                   = false
```

In a unit test written with `check-expect`, lists are compared for equality structurally, the way the μScheme function `equal?` does.

S380d. ⟨*utility functions on values (μScheme, Typed μScheme, nano-ML)* S380c⟩+≡ (S379) ◁ S380c S380e ▷

```
equalpairs : value * value -> bool
```

```
fun equalpairs (PAIR (car1, cdr1), PAIR (car2, cdr2)) =
      equalpairs (car1, car2) andalso equalpairs (cdr1, cdr2)
  | equalpairs (v1, v2) = equalatoms (v1, v2)
```

The testing infrastructure expects this function to be called `testEquals`.

S380e. ⟨*utility functions on values (μScheme, Typed μScheme, nano-ML)* S380c⟩+≡ (S379) ◁ S380d S389c ▷

```
val testEquals = equalpairs
```

```
testEquals : value * value -> bool
```

O.2 Primitive functions and the initial basis

O.2.1 *Defining the remaining primitives*

Some primitives are defined in Chapter 5. The rest are here.

S380f. ⟨*primitives for μScheme* `::` S380f⟩≡ (S382a) S381a ▷

```
("number?",  predOp (fn (NUM   _) => true | _ => false)) ::
("symbol?",  predOp (fn (SYM   _) => true | _ => false)) ::
("pair?",    predOp (fn (PAIR  _) => true | _ => false)) ::
("function?",
      predOp (fn (PRIMITIVE _) => true
               | (CLOSURE   _) => true
               | _ => false)) ::
```

The list primitives are also implemented by simple anonymous functions:

S381a. ⟨*primitives for μScheme* :: S380f⟩+≡ (S382a) ◁S380f S381b▷

```
("cons", binaryOp (fn (a, b) => PAIR (a, b))) ::
("car",  unaryOp  (fn (PAIR (car, _)) => car
                    | NIL => raise RuntimeError "car applied to empty list"
                    | v => raise RuntimeError
                             ("car applied to non-list " ^ valueString v))) ::
("cdr",  unaryOp  (fn (PAIR (_, cdr)) => cdr
                    | NIL => raise RuntimeError "cdr applied to empty list"
                    | v => raise RuntimeError
                             ("cdr applied to non-list " ^ valueString v))) ::
```

The ML-version of μScheme includes a secret hash primitive.

S381b. ⟨*primitives for μScheme* :: S380f⟩+≡ (S382a) ◁S381a S381c▷

```
("hash",  unaryOp (fn SYM s => NUM (fnvHash s)
                    | v => raise RuntimeError
                              (valueString v ^ " is not a symbol"))) ::
```

The printing primitives are all similar.

S381c. ⟨*primitives for μScheme* :: S380f⟩+≡ (S382a) ◁S381b

```
("println", unaryOp (fn v => (print (valueString v ^ "\n"); v))) ::
("print",   unaryOp (fn v => (print (valueString v);        v))) ::
("printu",  unaryOp (fn NUM n => (printUTF8 n; NUM n)
                      | v => raise RuntimeError
                               (valueString v ^
                                " is not a Unicode code point"))) ::
```

The primitives in the list defined by ⟨*primitives for μScheme* :: S380f⟩ all have type value list -> value. They are lifted into actual primitives by function inExp (chunk 313a), which extends each run-time error message with a rendering of the expression whose evaluation caused the error. But for the error primitive, extending the message is not usually appropriate.

Although error raises the RuntimeError exception, raising the exception is expected behavior, and the context in which the exception is raised should not be shown—unless error is given the wrong number of arguments. To maintain such fine control over its behavior, errorPrimitive takes an exp parameter on its own, and it delegates reporting to inExp only in the case of an arity error.

S381d. ⟨*utility functions for building primitives in μScheme* S381d⟩≡ (S382a)

```
errorPrimitive : exp * value list -> value list
```

```
fun errorPrimitive (_, [v]) = raise RuntimeError (valueString v)
  | errorPrimitive (e, vs)  = inExp (arityError 1) (e, vs)
```

O.2.2 Using primitives to build an initial basis

A basis for μScheme comprises a single value environment. The initial basis is built by starting with the empty environment, binding the primitive operators, then reading the predefined functions.

All the primitives on the list defined by ⟨*primitives for μScheme* :: S380f⟩ are lifted using `inExp`. Only the `errorPrimitive` is bound straight into the environment.

S382a. ⟨*implementations of μScheme primitives and definition of* `initialBasis` S382a⟩≡ (S379) S382b ▷

```
primitiveBasis : basis
initialBasis : basis
```

```
⟨utility functions for building primitives in μScheme S381d⟩
val primitiveBasis =
  let val rho =
        foldl (fn ((name, prim), rho) =>
                 bind (name, ref (PRIMITIVE (inExp prim)), rho))
              emptyEnv
              (⟨primitives for μScheme :: S380f⟩ [])
      val rho = bind ("error", ref (PRIMITIVE errorPrimitive), rho)
  in  rho
  end
```

When reading predefined functions, the interpreter echoes no responses, and to issue error messages, it uses the special function `predefinedError`.

S382b. ⟨*implementations of μScheme primitives and definition of* `initialBasis` S382a⟩+≡ (S379) ◁ S382a

```
val predefs = ⟨predefined μScheme functions, as strings (from ⟨additions to the μScheme initial basis 96a⟩)⟩

val initialBasis =
  let val xdefs = stringsxdefs ("predefined functions", predefs)
  in  readEvalPrintWith predefinedFunctionError
                        (xdefs, primitiveBasis, noninteractive)
  end
```

O.3 Unit tests for μScheme

Interpreters that are written in ML use a single language-dependent testing function, called `testIsGood`. Unlike the corresponding C function, `test_result`, `testIsGood` returns a Boolean. That's because the code is simple enough, and it uses enough named auxiliary functions—functions `passes`, `checkExpectPasses`, `checkAssertPasses`, and `checkErrorPasses`—that the meaning of the Boolean is always clear from context. You might enjoy comparing the code below with the C code on pages S301 to S303, which returns a value of enumeration type, not a Boolean. The C code is so complicated that I *don't* know from context what a Boolean result is supposed to mean; that's why I define and use the enumeration type `TestResult` on page S300.

In μScheme, a test is good if it passes. (In some other languages, a good test must also be well typed.) The "pass" functions themselves are defined in Appendix H.

S382c. ⟨*definition of* `testIsGood` *for μScheme* S382c⟩≡ (S380b)

```
testIsGood : unit_test * basis -> bool
outcome    : exp -> value error
```

```
fun testIsGood (test, rho) =
  let fun outcome e =
        withHandlers (fn e => OK (eval (e, rho))) e (ERROR o stripAtLoc)
      ⟨asSyntacticValue for μScheme, Typed Impcore, Typed μScheme, and nano-ML S383a⟩
      ⟨shared check{Expect,Assert,Error{Passes, which call outcome S224d⟩
      fun passes (CHECK_EXPECT (c, e)) = checkExpectPasses (c, e)
        | passes (CHECK_ASSERT c)      = checkAssertPasses c
        | passes (CHECK_ERROR c)       = checkErrorPasses  c
  in  passes test
  end
```

In most languages, only literal expressions are considered syntactic values.

S383a. ⟨asSyntacticValue *for μScheme, Typed Impcore, Typed μScheme, and nano-ML* S383a⟩≡ (S382c)

```
asSyntacticValue : exp -> value option
```

```
fun asSyntacticValue (LITERAL v) = SOME v
  | asSyntacticValue _           = NONE
```

When a test fails, the offending expression is printed. Abstract syntax is converted to a string by function `expString`.

S383b. ⟨*definition of* expString *for μScheme* S383b⟩≡ (S380a)

```
fun expString e =
  let fun bracket s = "(" ^ s ^ ")"
      val bracketSpace = bracket o spaceSep
      fun exps es = map expString es
      fun withBindings (keyword, bs, e) =
        bracket (spaceSep [keyword, bindings bs, expString e])
      and bindings bs = bracket (spaceSep (map binding bs))
      and binding (x, e) = bracket (x ^ " " ^ expString e)
      val letkind = fn LET => "let" | LETSTAR => "let*" | LETREC => "letrec"
  in  case e
        of LITERAL (v as NUM   _) => valueString v
         | LITERAL (v as BOOLV _) => valueString v
         | LITERAL v => "'" ^ valueString v
         | VAR name => name
         | SET (x, e) => bracketSpace ["set", x, expString e]
         | IFX (e1, e2, e3) => bracketSpace ("if" :: exps [e1, e2, e3])
         | WHILEX (cond, body) =>
                          bracketSpace ["while", expString cond, expString body]
         | BEGIN es => bracketSpace ("begin" :: exps es)
         | APPLY (e, es) => bracketSpace (exps (e::es))
         | LETX (lk, bs, e) => bracketSpace [letkind lk, bindings bs, expString
         | LAMBDA (xs, body) => bracketSpace ["lambda", bracketSpace xs, expStri
  end
```

O.4 Lexical analysis and parsing

Lexical analysis and parsing is implemented by these code chunks:

S383c. ⟨*lexical analysis and parsing for μScheme, providing* filexdefs *and* stringsxdefs S383c⟩≡

⟨*lexical analysis for μScheme and related languages* S383d⟩
⟨*parsers for single tokens for μScheme-like languages* S385a⟩
⟨*parsers for μScheme tokens* S385b⟩
⟨*parsers and parser builders for formal parameters and bindings* S385c⟩
⟨*parsers and parser builders for Scheme-like syntax* S386d⟩
⟨*parsers and* xdef *streams for μScheme* S387b⟩
⟨*shared definitions of* filexdefs *and* stringsxdefs S233a⟩

O.4.1 *Tokens of the μScheme language*

The general parsing mechanism described in Appendix I requires a language-specific `pretoken` type, which adds tokens to the brackets described in Appendix I. In addition to a bracket, a μScheme token may be a quote mark, an integer literal, a Boolean literal, or a name.

S383d. ⟨*lexical analysis for μScheme and related languages* S383d⟩≡ (S383c) S384a ▷

```
type pretoken
type token
```

```
datatype pretoken = QUOTE
                  | INT     of int
                  | SHARP   of bool
                  | NAME    of string
type token = pretoken plus_brackets
```

For debugging, code in Appendix I needs to be able to render a `token` as a string.

S384a. ⟨*lexical analysis for μScheme and related languages* S383d⟩+≡ (S383c) ◁S383d S384b▷

```
pretokenString : pretoken -> string
tokenString    : token    -> string
```

```
fun pretokenString (QUOTE)     = "'"
  | pretokenString (INT  n)    = intString n
  | pretokenString (SHARP b)   = if b then "#t" else "#f"
  | pretokenString (NAME x)    = x
val tokenString = plusBracketsString pretokenString
```

O.4.2 *Lexical analysis for μScheme*

Lexical analysis turns a character stream into a token stream. The `schemeToken` function tries each alternative in turn: the two brackets, a quote mark, an integer literal, an atom, or end of line. An atom may be a `SHARP` name or a normal name.

Before each μScheme token, whitespace is ignored.

S384b. ⟨*lexical analysis for μScheme and related languages* S383d⟩+≡ (S383c) ◁S384a

```
schemeToken : token lexer
atom : string -> pretoken
```

```
local
  ⟨functions used in all lexers S384d⟩
  ⟨functions used in the lexer for μScheme S384c⟩
in
  val schemeToken =
    whitespace *>
    bracketLexer   (  QUOTE   <$  eqx #"'" one
                  <|> INT     <$> intToken isDelim
                  <|> (atom o implode) <$> many1 (sat (not o isDelim) one)
                  <|> noneIfLineEnds
                   )
end
```

The `atom` function identifies the special literals `#t` and `#f`; all other atoms are names.

S384c. ⟨*functions used in the lexer for μScheme* S384c⟩≡ (S384b)

```
fun atom "#t" = SHARP true
  | atom "#f" = SHARP false
  | atom x    = NAME x
```

If the lexer doesn't recognize a bracket, quote mark, integer, or other atom, it's expecting the line to end. The end of the line may present itself as the end of the input stream or as a stream of characters beginning with a semicolon, which marks a comment. If the lexer encounters any other character, something has gone wrong. (The polymorphic type of `noneIfLineEnds` provides a subtle but strong hint that no token can be produced; the only possible outcomes are that nothing is produced, or the lexer detects an error.)

S384d. ⟨*functions used in all lexers* S384d⟩≡ (S384b)

```
noneIfLineEnds : 'a lexer
```

```
fun noneIfLineEnds chars =
  case streamGet chars
    of NONE => NONE (* end of line *)
     | SOME (#";", cs) => NONE (* comment *)
     | SOME (c, cs) =>
         let val msg = "invalid initial character in `" ^
                       implode (c::listOfStream cs) ^ "'"
         in  SOME (ERROR msg, EOS)
         end
```

O.4.3 Parsers for μScheme

A parser consumes a stream of tokens and produces an abstract-syntax tree. My parsers begin with code for parsing the smallest things and finish with the code for parsing the biggest things. I define parsers for tokens, literal S-expressions, μScheme expressions, and finally μScheme definitions.

Parsers for single tokens and common idioms

Usually a parser knows what kind of token it is looking for. To make such a parser easier to write, I define a special parsing combinator for each kind of token. Each one succeeds when given a token of the kind it expects; when given any other token, it fails.

S385a. ⟨*parsers for single tokens for μScheme-like languages* S385a⟩≡ (S383c)

```
booltok  : bool parser
int      : int  parser
namelike : name parser
```

```
type 'a parser = (token, 'a) polyparser
val pretoken  = (fn (PRETOKEN t)=> SOME t  | _ => NONE) <$>? token : pretoken parser
val quote     = (fn (QUOTE)     => SOME () | _ => NONE) <$>? pretoken
val int       = (fn (INT   n)   => SOME n  | _ => NONE) <$>? pretoken
val booltok   = (fn (SHARP b)   => SOME b  | _ => NONE) <$>? pretoken
val namelike  = (fn (NAME  n)   => SOME n  | _ => NONE) <$>? pretoken
val namelike  = asAscii namelike
```

A `namelike` parser accepts any name, as long as the name is made up only of printing ASCII characters. But it also accepts reserved words, and reserved words must not be accepted as names.

S385b. ⟨*parsers for μScheme tokens* S385b⟩≡ (S383c)

```
val reserved = [ "if", "while", "set", "begin", "lambda", "let"
               , "letrec", "let*", "quote", "val", "define", "use"
               , "check-expect", "check-assert", "check-error"
               ]
val name = rejectReserved reserved <$>! namelike
```

Building on the single tokens, I define parsers that handle syntactic elements used in multiple Scheme-like languages. Function `formals` parses a list of formal parameters. In a list of formal parameters, if not all parameter names are mutually distinct, it's treated as a syntax error. Function `bindings` produces a list of bindings suitable for use in `let*` expressions.

S385c. ⟨*parsers and parser builders for formal parameters and bindings* S385c⟩≡ (S383c) S386a ▷

```
formalsOf  : string -> name parser -> string -> name list parser
bindingsOf : string -> 'x parser -> 'e parser -> ('x * 'e) list parser
```

```
fun formalsOf what name context =
  nodups ("formal parameter", context) <$>! @@ (bracket (what, many name))

fun bindingsOf what name exp =
  let val binding = bracket (what, pair <$> name <*> exp)
  in  bracket ("(... " ^ what ^ " ...) in bindings", many binding)
  end
```

For `let` and `letrec` expressions, which do not permit multiple bindings to the same name, I define function `distinctBsIn`, which enforces the constraint that all bound names are mutually distinct.

S386a. ⟨*parsers and parser builders for formal parameters and bindings* S385c⟩+≡ (S383c) ◁S385c S386c▷

```
distinctBsIn : (name * 'e) list parser -> string -> (name * 'e) list parser
```

```
fun distinctBsIn bindings context =
  let fun check (loc, bs) =
        nodups ("bound name", context) (loc, map fst bs) >>=+ (fn _ => bs)
  in  check <$>! @@ bindings
  end
```

A `letrec` may bind only lambda expressions.

S386b. ⟨*parsers and parser builders for formal parameters and bindings* **⟦higher-order⟧** S386b⟩≡

```
asLambda : string -> exp parser -> exp parser
```

```
fun asLambda inWhat (loc, e as LAMBDA _) = OK e
  | asLambda inWhat (loc, e) =
      synerrorAt ("in " ^ inWhat ^ ", expression " ^ expString e ^
                  " is not a lambda")
                 loc

val asLambda = fn what => fn eparser => asLambda what <$>! @@ eparser
```

Record fields must also have mutually distinct names.

S386c. ⟨*parsers and parser builders for formal parameters and bindings* S385c⟩+≡ (S383c) ◁S386a S387a▷

```
recordFieldsOf : name parser -> name list parser
```

```
fun recordFieldsOf name =
  nodups ("record fields", "record definition") <$>!
                                  @@ (bracket ("(field ...)", many name))
```

With this machinery I can define a parser for quoted S-expressions. A quoted S-expression is a symbol, a number, a Boolean, a list of S-expressions, or a quoted S-expression.

S386d. ⟨*parsers and parser builders for Scheme-like syntax* S386d⟩≡ (S383c) S386e▷

```
sexp : value parser
```

```
fun sexp tokens = (
     SYM       <$> (notDot <$>! @@ namelike)
 <|> NUM       <$> int
 <|> embedBool <$> booltok
 <|> leftCurly <!> "curly brackets may not be used in S-expressions"
 <|> embedList <$> bracket ("list of S-expressions", many sexp)
 <|> (fn v => embedList [SYM "quote", v])
               <$> (quote *> sexp)
) tokens
and notDot (loc, ".") =
      synerrorAt "this interpreter cannot handle . in quoted S-expressions" loc
  | notDot (_,   s)   = OK s
```

Full Scheme allows programmers to notate arbitrary cons cells using a dot in a quoted S-expression. μScheme doesn't.

Parsers for μScheme expressions

I define distinct parses for atomic expressions (which aren't recursively defined) and bracketed expressions (which are recursively defined). An atomic expression is a variable or a literal.

S386e. ⟨*parsers and parser builders for Scheme-like syntax* S386d⟩+≡ (S383c) ◁S386d S388a▷

```
fun atomicSchemeExpOf name =  VAR                   <$> name
                          <|> LITERAL <$> NUM       <$> int
                          <|> LITERAL <$> embedBool <$> booltok
```

To define a parser for the bracketed expressions, I deploy the “usage parser” described in Appendix I. It enables me to define most of the parser as a table containing usage strings and functions. Function `kw` parses only the keyword passed as an argument. Using it, function `usageParsers` strings together usage parsers.

S387a. ⟨*parsers and parser builders for formal parameters and bindings* S385c⟩+≡ (S383c) ◁S386c

```
kw : string -> string parser
usageParsers : (string * 'a parser) list -> 'a parser
```

```
fun kw keyword =
  eqx keyword namelike
fun usageParsers ps = anyParser (map (usageParser kw) ps)
```

Function `exptable`, when given a parser `exp` for all expressions, produces a parser for bracketed expressions. In the C code in Appendix L the data structure `exptable` is mutually recursive with functions `parseexp`, `sExp`, and `reduce_to_exp`. In ML, such mutual recursion is difficult to achieve. The technique I use here is to define `exptable` as a function, which is passed function `exp` as a parameter. Below, recursive function `exp` is defined to use both itself and `exptable`.

The `exptable` itself uses the format described in Section I.3.4 (page S264): each alternative is specified by a pair containing a usage string and a parser.

S387b. ⟨*parsers and* `xdef` *streams for μScheme* S387b⟩≡ (S383c) S388b ▷

```
exptable  : exp parser -> exp parser
exp       : exp parser
bindings  : (name * exp) list parser
```

```
fun exptable exp =
  let val bindings = bindingsOf "(x e)" name exp
      val formals  = formalsOf "(x1 x2 ...)" name "lambda"
      val dbs      = distinctBsIn bindings
      val letrecbs =
        distinctBsIn
            (bindingsOf "[f (lambda (...) ...)]" name (asLambda "letrec" exp))
            "letrec"
  in usageParsers
     [ ("(if e1 e2 e3)",           curry3 IFX    <$> exp <*> exp <*> exp)
     , ("(while e1 e2)",           curry  WHILEX <$> exp <*> exp)
     , ("(set x e)",               curry  SET    <$> name <*> exp)
     , ("(begin e1 ...)",                 BEGIN  <$> many exp)
     , ("(lambda (names) body)",   curry  LAMBDA <$> formals <*> exp)
     , ("(let (bindings) body)",   curry3 LETX LET     <$> dbs  "let" <*> exp)
     , ("(letrec (bindings) body)", curry3 LETX LETREC  <$> letrecbs   <*> exp)
     , ("(let* (bindings) body)",  curry3 LETX LETSTAR <$> bindings   <*> exp)
     , ("(quote sexp)",            LITERAL             <$> sexp)
     ⟨rows added to ML μScheme's exptable in exercises S387c⟩
     ]
  end
```

To support some of the exercises in the main text, the list of parsers includes a placeholder where more parsers can be added.

S387c. ⟨*rows added to ML μScheme's* `exptable` *in exercises* S387c⟩≡ (S387b)

```
(* add syntactic sugar here, each row preceded by a comma *)
```

The full expression parser handles atomic expressions, quoted S-expressions, the table of bracketed expressions, a couple of error cases, and function application, which uses parentheses but no keyword. It is parameterized by the parsers for atomic expressions and bracketed expressions. This parameterization enables me to reuse it in other interpreters.

S388a. ⟨*parsers and parser builders for Scheme-like syntax* S386d⟩+≡ (S383c) ◁S386e

```
fullSchemeExpOf : exp parser -> (exp parser -> exp parser) -> exp parser
```

```
fun fullSchemeExpOf atomic bracketedOf =
  let val exp = fn tokens => fullSchemeExpOf atomic bracketedOf tokens
  in      atomic
      <|> bracketedOf exp
      <|> quote *> (LITERAL <$> sexp)
      <|> quote *> badRight "quote ' followed by right bracket"
      <|> leftCurly <!> "curly brackets are not supported"
      <|> left *> right <!> "(): unquoted empty parentheses"
      <|> bracket("function application", curry APPLY <$> exp <*> many exp)
  end
```

The actual `exp` parser is defined by instantiating the general one with the correct parsers for atomic and bracketed expressions.

S388b. ⟨*parsers and* `xdef` *streams for μScheme* S387b⟩+≡ (S383c) ◁S387b S388c▷

```
val exp = fullSchemeExpOf (atomicSchemeExpOf name) exptable
```

Parsers for μScheme definitions

I segregate the definition parsers by the ML type of definition they produce. Parser `deftable` parses the true definitions.

S388c. ⟨*parsers and* `xdef` *streams for μScheme* S387b⟩+≡ (S383c) ◁S388b S388d▷

```
deftable : def parser
```

```
val deftable = usageParsers
  [ ("(define f (args) body)",
        let val formals  = formalsOf "(x1 x2 ...)" name "define"
        in  curry DEFINE <$> name <*> (pair <$> formals <*> exp)
        end)
  , ("(val x e)", curry VAL <$> name <*> exp)
  ]
```

Parser `testtable` parses the unit tests.

S388d. ⟨*parsers and* `xdef` *streams for μScheme* S387b⟩+≡ (S383c) ◁S388c S388e▷

```
testtable : unit_test parser
```

```
val testtable = usageParsers
  [ ("(check-expect e1 e2)", curry CHECK_EXPECT <$> exp <*> exp)
  , ("(check-assert e)",           CHECK_ASSERT <$> exp)
  , ("(check-error e)",            CHECK_ERROR  <$> exp)
  ]
```

Parser `xdeftable` handles those extended definitions that are not unit tests. It is also where you would extend the parser with new syntactic forms of definition, like the `record` form described in Section 2.13.6 (page 167).

S388e. ⟨*parsers and* `xdef` *streams for μScheme* S387b⟩+≡ (S383c) ◁S388d S389a▷

```
xdeftable : xdef parser
```

```
val xdeftable = usageParsers
  [ ("(use filename)", USE <$> name)
  ⟨rows added to μScheme xdeftable in exercises S388f⟩
  ]
```

S388f. ⟨*rows added to μScheme* `xdeftable` *in exercises* S388f⟩≡ (S388e)

```
(* add syntactic sugar here, each row preceded by a comma *)
```

The `xdef` parser combines all the types of extended definition, plus an error case.

S389a. ⟨*parsers and* `xdef` *streams for μScheme* S387b⟩+≡ (S383c) ◁S388e S389b▷

```
xdef : xdef parser
```

```
val xdef =  DEF  <$> deftable
        <|> TEST <$> testtable
        <|>          xdeftable
        <|> badRight "unexpected right bracket"
        <|> DEF <$> EXP <$> exp
        <?> "definition"
```

Finally, function `xdefstream`, which is the externally visible interface to the parsing, uses the lexer and parser to make a function that converts a stream of lines to a stream of extended definitions.

S389b. ⟨*parsers and* `xdef` *streams for μScheme* S387b⟩+≡ (S383c) ◁S389a

```
xdefstream : string * line stream * prompts -> xdef stream
```

```
val xdefstream =
  interactiveParsedStream (schemeToken, xdef)
```

O.5 Unspecified values

In a `val` or `letrec` binding, the operational semantics of μScheme call for the allocation of a location containing an unspecified value. My C code chooses a value at random, but the initial basis of Standard ML has no random-number generator. So unlike the C `unspecified` function in chunk S328a, the ML version just cycles through a few different values. It's probably enough to prevent careless people from assuming that such a value is always `NIL`.

S389c. ⟨*utility functions on values (μScheme, Typed μScheme, nano-ML)* S380c⟩+≡ (S379) ◁S380e

```
cycleThrough : 'a list -> (unit -> 'a)
unspecified  : unit -> value
```

```
fun cycleThrough xs =
  let val remaining = ref xs
      fun next () = case !remaining
                      of [] => (remaining := xs; next ())
                       | x :: xs => (remaining := xs; x)
  in  if null xs then
        raise InternalError "empty list given to cycleThrough"
      else
        next
  end
val unspecified =
  cycleThrough [ BOOLV true, NUM 39, SYM "this value is unspecified", NIL
               , PRIMITIVE (fn _ => raise RuntimeError "unspecified primitive")
               ]
```

O.6 Further reading

Koenig (1994) describes an experience with ML type inference which leads to a conclusion that resembles my conclusion about the type of `noneIfLineEnds` on page S384d.

`<!>`	S261a
`<$>`	S249a
`<*>`	S247b
`<?>`	S260c
`<|>`	S249c
`APPLY`	306
`atomicSchemeExpOf`	S386e
`badRight`	S262a
`BOOLV`	306
`bracket`	S264a
`CHECK_ASSERT`	S214a
`CHECK_ERROR`	S214a
`CHECK_EXPECT`	S214a
`curry`	S249b
`DEF`	S214b
`type def`	307a
`DEFINE`	307a
`EXP`	307a
`type exp`	306
`exptable`	S387b
`formalsOf`	S385c
`interactiveParsedStream`	S269
`InternalError`	S219e
`left`	S261c
`leftCurly`	S262b
`LITERAL`	306
`many`	S253a
`name`	S385b
`NIL`	306
`NUM`	306
`pair`	S249b
`type parser`	S385a
`PRIMITIVE`	306
`quote`	S385a
`right`	S261c
`RuntimeError`	S213b
`schemeToken`	S384b
`sexp`	S386d
`type stream`	S228c
`SYM`	306
`TEST`	S214b
`type unit_test`	S214a
`usageParsers`	S387a
`USE`	S214b
`VAL`	307a
`type value`	306
`type xdef`	S214b

Appendix P contents

P

Supporting code for Typed Impcore

P.1 Organizing code chunks into an interpreter

Like all the interpreters from Chapter 5 onward, the Typed Impcore interpreter is defined by laying down Noweb chunks in the right order, as discussed in Appendix O. The layout is similar to that of μScheme, but Typed Impcore has two additional chunks which are related to type checking: ⟨*types for Typed Impcore* 331f⟩ and ⟨*type checking for Typed Impcore* 338a⟩.

S391a. ⟨*timpcore.sml* S391a⟩≡

```
  ⟨exceptions used in languages with type checking S213c⟩
  ⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization S213a⟩

  ⟨types for Typed Impcore 331f⟩

  ⟨abstract syntax and values for Typed Impcore S391b⟩
  ⟨utility functions on Typed Impcore values S394d⟩

  ⟨type checking for Typed Impcore 338a⟩

  ⟨lexical analysis and parsing for Typed Impcore, providing filexdefs and stringsxdefs S399c⟩

  ⟨evaluation, testing, and the read-eval-print loop for Typed Impcore S396e⟩

  ⟨implementations of Typed Impcore primitives and definition of initialBasis S394c⟩
  ⟨function runStream, which evaluates input given initialBasis S240b⟩
  ⟨look at command-line arguments, then run S240c⟩
```

Chunks related to abstract syntax are collected here. The definition of `xdef` is shared with μScheme, and functions `valueString` and `expString` are defined below.

S391b. ⟨*abstract syntax and values for Typed Impcore* S391b⟩≡ (S391a)

```
  ⟨definitions of exp and value for Typed Impcore 332c⟩
  ⟨definition of type func, to represent a Typed Impcore function 333c⟩
  ⟨definition of def for Typed Impcore 333a⟩
  ⟨definition of unit_test for Typed Impcore 333b⟩
  ⟨definition of xdef (shared) S214b⟩
  ⟨definition of valueString for Typed Impcore S398d⟩
  ⟨definition of expString for Typed Impcore S399a⟩
  ⟨definitions of defString and defName for Typed Impcore S399b⟩
  ⟨definitions of functions toArray and toInt for Typed Impcore 345a⟩
```

For Typed Impcore to work with the reusable read-eval-print loop described in Section H.7 (page S235), I need to define the type `basis` and function `processDef`.

The `processDef` function for a dynamically typed language such as Impcore or μScheme can simply evaluate a definition. But the `processDef` function for a statically typed language such as Typed Impcore also needs a typechecking step.

Function `processDef` needs not only the top-level type environments Γ_ϕ and Γ_ξ but also the top-level value and function environments ϕ and ξ. These environments are put into a tuple whose type is `basis`. Of the four environments, the value environment ξ is the only one that can be mutated during evaluation, so it is the only one that has a `ref` in its type.

S392a. ⟨*definitions of* `basis` *and* `processDef` *for Typed Impcore* S392a⟩≡ (S396e)

```
type basis
processDef : def * basis * interactivity -> basis
```

```
type basis = ty env * funty env * value ref env * func env
fun processDef (d, (tglobals, tfuns, vglobals, vfuns), interactivity) =
  let val (tglobals, tfuns, tystring)  = typdef  (d, tglobals, tfuns)
      val (vglobals, vfuns, valstring) = evaldef (d, vglobals, vfuns)
      val _ = if echoes interactivity then println (valstring ^ " : " ^ tystring)
              else ()
  in  (tglobals, tfuns, vglobals, vfuns)
  end
```

The distinction between "compile time," where the interpreter runs the typing phase `typdef`, and "run time," where the interpreter runs the evaluator `evaldef`, is sometimes called the *phase distinction*. The phase distinction is easy to overlook, especially when you're using an interactive interpreter or compiler, but the code shows the phase distinction is real.

Function `typdef` is defined in Chapter 6, and function `evaldef` is defined below. But one part of `typdef` is defined here: function `badParameter` is called when a function application fails to typecheck, in which case `badParameter` formulates an informative error message.

S392b. ⟨*definition of* `badParameter` S392b⟩≡ (341a)

```
badParameter : int * ty list * ty list -> 'a
```

```
fun badParameter (n, atau::actuals, ftau::formals) =
      if eqType (atau, ftau) then
        badParameter (n+1, actuals, formals)
      else
        raise TypeError ("In call to " ^ f ^ ", parameter " ^
                         intString n ^ " has type " ^ typeString atau ^
                         " where type " ^ typeString ftau ^ " is expected")
  | badParameter _ =
      raise TypeError ("Function " ^ f ^ " expects " ^
                       countString formaltypes "parameter" ^
                       " but got " ^ intString (length actualtypes))
```

Because `badParameter` is called only when the code fails to typecheck, it doesn't need for a base case in which both lists are empty.

Function `processDef` is used in the read-eval-print loop that is defined in Section H.7 (page S235). The code for the loop can be used unchanged except for one addition: Typed Impcore needs handlers for the new exceptions introduced in Chapter 6 (`TypeError` and `BugInTypeChecking`). `TypeError` is raised not at parsing time, and not at evaluation time, but at typechecking time (by `typdef`). `BugInTypeChecking` should never be raised.

S392c. ⟨*other handlers that catch non-fatal exceptions and pass messages to* `caught` S392c⟩≡

```
| TypeError         msg => caught ("type error <at loc>: " ^ msg)
| BugInTypeChecking msg => caught ("bug in type checking: " ^ msg)
```

S392d. ⟨*more handlers for* `atLoc` S392d⟩≡

```
| e as TypeError _         => raise Located (loc, e)
| e as BugInTypeChecking _ => raise Located (loc, e)
```

P.2 Primitive functions, predefined functions, and the initial basis

Code in this section defines Typed Impcore's primitive functions and builds its initial basis. As in Chapter 5, all primitives are either binary or unary operators. But the code below should not reuse functions `unaryOp` and `binaryOp` from Chapter 5, because when a primitive is called with the wrong number of arguments, those versions raise the `RuntimeError` exception. In a typed language, it should not be possible to call a primitive with the wrong number of arguments. If it happens anyway, these versions of `unaryOp` and `binaryOp` raise `BugInTypeChecking`.

S393a. ⟨*functions for building primitives when types are checked* S393a⟩≡ (S394c S406d) S393b ▷

```
unaryOp  : (value         -> value) -> (value list -> value)
binaryOp : (value * value -> value) -> (value list -> value)
```

```
fun binaryOp f = (fn [a, b] => f (a, b) | _ => raise BugInTypeChecking "arity 2")
fun unaryOp  f = (fn [a]    => f a      | _ => raise BugInTypeChecking "arity 1")
```

Arithmetic primitives expect and return integers.

S393b. ⟨*functions for building primitives when types are checked* S393a⟩+≡ (S394c S406d) ◁ S393a

```
arithOp : (int * int -> int) -> (value list -> value)
```

```
fun arithOp f =
      binaryOp (fn (NUM n1, NUM n2) => NUM (f (n1, n2))
                 | _ => raise BugInTypeChecking "arithmetic on non-numbers")
```

S393c. ⟨*utilities used to make Typed Impcore primitives* S393c⟩≡ (S394c) S393e ▷

```
arithtype : funty
```

```
val arithtype = FUNTY ([INTTY, INTTY], INTTY)
```

As in Chapter 5, I use the chunk ⟨*primitive functions for Typed Impcore* `::` S393d⟩ to cons up all the primitives into one giant list, and that list is used to build the initial environment for the read-eval-print loop. What's new here is that in Typed Impcore, each primitive has a type as well as a value.

S393d. ⟨*primitive functions for Typed Impcore* `::` S393d⟩≡ (S394c) S393f ▷

```
("+", arithOp op +,   arithtype) ::
("-", arithOp op -,   arithtype) ::
("*", arithOp op *,   arithtype) ::
("/", arithOp op div, arithtype) ::
```

Comparisons take two arguments. Most comparisons (except for equality) apply only to integers.

S393e. ⟨*utilities used to make Typed Impcore primitives* S393c⟩+≡ (S394c) ◁ S393c

```
comparison : (value * value -> bool) -> (value list -> value)
intcompare : (int   * int   -> bool) -> (value list -> value)
comptype   : funty
```

```
fun embedBool  b = NUM (if b then 1 else 0)
fun comparison f = binaryOp (embedBool o f)
fun intcompare f =
      comparison (fn (NUM n1, NUM n2) => f (n1, n2)
                   | _ => raise BugInTypeChecking "comparing non-numbers")
val comptype = FUNTY ([INTTY, INTTY], BOOLTY)
```

S393f. ⟨*primitive functions for Typed Impcore* `::` S393d⟩+≡ (S394c) ◁ S393d S394a ▷

```
("<", intcompare op <, comptype) ::
(">", intcompare op >, comptype) ::
```

Typed Impcore also has a primitive that prints Unicode. (Unlike `print`, `printu` works only on integer arguments, so unlike `print` it can be implemented as a primitive function rather than a syntactic form.)

S394a. ⟨*primitive functions for Typed Impcore* **::** S393d⟩+≡ (S394c) ◁S393f

```
("printu", unaryOp (fn (NUM n) => (printUTF8 n; unitVal)
                     | _ => raise BugInTypeChecking "printu of non-number"),
                FUNTY ([INTTY], UNITTY)) ::
```

Like Impcore, Typed Impcore predefines only about half a dozen functions. Most are defined in Chapter 6, but modulus and negation are defined here.

S394b. ⟨*predefined Typed Impcore functions* S394b⟩≡

```
(define int mod ([m : int] [n : int]) (- m (* n (/ m n))))
(define int negated ([n : int]) (- 0 n))
```

The initial basis includes both primitive and predefined functions.

S394c. ⟨*implementations of Typed Impcore primitives and definition of* `initialBasis` S394c⟩≡ (S391a)

```
⟨functions for building primitives when types are checked S393a⟩
⟨utilities used to make Typed Impcore primitives S393c⟩
val primitiveBasis : basis =
  let fun addPrim ((name, prim, funty), (tfuns, vfuns)) =
        ( bind (name, funty, tfuns)
        , bind (name, PRIMITIVE prim, vfuns)
        )
      val (tfuns, vfuns)  = foldl addPrim (emptyEnv, emptyEnv)
                            (⟨primitive functions for Typed Impcore :: S393d⟩ nil)
  in  (emptyEnv, tfuns, emptyEnv, vfuns)
  end

val predefs = ⟨predefined Typed Impcore functions, as strings (from chunk 331d)⟩

val initialBasis =
  let val xdefs = stringsxdefs ("predefined functions", predefs)
  in  readEvalPrintWith
        predefinedFunctionError
        (xdefs, primitiveBasis, noninteractive)
  end
```

P.3 Unit testing

In Typed Impcore, numbers can equal numbers and arrays can equal arrays. Because arrays are mutable, Typed Impcore defines equality as object identity, which is implemented using ML's polymorphic = form

S394d. ⟨*utility functions on Typed Impcore values* S394d⟩≡ (S391a)

```
fun testEquals (NUM n,   NUM n')   = n = n'
  | testEquals (ARRAY a, ARRAY a') = a = a'
  | testEquals (_,       _)        = false
```

In Typed Impcore a good unit test has to typecheck. Just passing an evaluation test isn't good enough.

S395a. ⟨*definition of* testIsGood *for Typed Impcore* S395a⟩≡ (S396e)

```
fun testIsGood (test, (tglobals, tfuns, vglobals, vfuns)) =
  let fun ty e = typeof (e, tglobals, tfuns, emptyEnv)
                 handle NotFound x =>
                   raise TypeError ("name " ^ x ^ " is not defined")
      fun deftystring d =
        let val (_, _, t) = typdef (d, tglobals, tfuns)
        in  t
        end handle NotFound x =>
              raise TypeError ("name " ^ x ^ " is not defined")
      ⟨shared check{Expect,Assert,Error,Type{Checks, which call ty S396c⟩
      fun checks (CHECK_EXPECT (e1, e2)) = checkExpectChecks (e1, e2)
        | checks (CHECK_ASSERT e)        = checkAssertChecks e
        | checks (CHECK_ERROR e)         = checkErrorChecks e
        | checks (CHECK_TYPE_ERROR d)    = true
        | checks (CHECK_FUNCTION_TYPE (f, fty)) = true

      fun outcome e =
        withHandlers (fn () => OK (eval (e, vglobals, vfuns, emptyEnv)))
                     ()
                     (ERROR o stripAtLoc)
      ⟨asSyntacticValue for μScheme, Typed Impcore, Typed μScheme, and nano-ML S383a⟩
      ⟨shared check{Expect,Assert,Error{Passes, which call outcome S224d⟩
      ⟨shared checkTypePasses and checkTypeErrorPasses, which call ty S396a⟩
      ⟨definition of checkFunctionTypePasses S395b⟩
      fun passes (CHECK_EXPECT (c, e))          = checkExpectPasses (c, e)
        | passes (CHECK_ASSERT c)               = checkAssertPasses c
        | passes (CHECK_ERROR c)                = checkErrorPasses c
        | passes (CHECK_FUNCTION_TYPE (f, fty)) = checkFunctionTypePasses (f, ft
        | passes (CHECK_TYPE_ERROR c)           = checkTypeErrorPasses c

  in  checks test andalso passes test
  end
```

S395b. ⟨*definition of* checkFunctionTypePasses S395b⟩≡ (S395a)

```
fun checkFunctionTypePasses (f, tau as FUNTY (args, result)) =
  let val tau' as FUNTY (args', result') =
            find (f, tfuns)
            handle NotFound f =>
              raise TypeError ("Function " ^ f ^ " is not defined")
  in  if eqTypes (args, args') andalso eqType (result, result') then
        true
      else
        failtest ["check-function-type failed: expected ", f,
                  " to have type ", funtyString tau,
                  ", but it has type ", funtyString tau']
  end handle TypeError msg =>
        failtest ["In (check-function-type ", f,
                  " " ^ funtyString tau, "), ", msg]
```

S396a. ⟨*shared* `checkTypePasses` *and* `checkTypeErrorPasses`*, which call* `ty` S396a⟩≡ (S395a S409a) S396b ▷

```
fun checkTypePasses (e, tau) =
  let val tau' = ty e
  in  if eqType (tau, tau') then
        true
      else
        failtest ["check-type failed: expected ", expString e,
                  " to have type ", typeString tau,
                  ", but it has type ", typeString tau']
  end handle TypeError msg =>
      failtest ["In (check-type ", expString e,
                " " ^ typeString tau, "), ", msg]
```

S396b. ⟨*shared* `checkTypePasses` *and* `checkTypeErrorPasses`*, which call* `ty` S396a⟩+≡ (S395a S409a) ◁ S396a

```
fun checkTypeErrorPasses (EXP e) =
      (let val tau = ty e
       in  failtest ["check-type-error failed: expected ", expString e,
                     " not to have a type, but it has type ", typeString tau]
       end handle TypeError msg => true
                | Located (_, TypeError _) => true)
  | checkTypeErrorPasses d =
      (let val t = deftystring d
       in  failtest ["check-type-error failed: expected ", defString d,
                         " to cause a type error, but it successfully defined ",
                         defName d, " : ", t
                        ]
       end handle TypeError msg => true
                | Located (_, TypeError _) => true)
```

S396c. ⟨*shared* `check{Expect,Assert,Error,Type{Checks`*, which call* `ty` S396c⟩≡ (S395a S409a) S396d ▷

```
fun checkExpectChecks (e1, e2) =
  let val tau1 = ty e1
      val tau2 = ty e2
  in  if eqType (tau1, tau2) then
        true
      else
        raise TypeError ("Expressions have types " ^ typeString tau1 ^
                            " and " ^ typeString tau2)
  end handle TypeError msg =>
  failtest ["In (check-expect ", expString e1, " ", expString e2, "), ", msg]
```

S396d. ⟨*shared* `check{Expect,Assert,Error,Type{Checks`*, which call* `ty` S396c⟩+≡ (S395a S409a) ◁ S396c

```
fun checkOneExpChecks inWhat e =
  let val tau1 = ty e
  in  true
  end handle TypeError msg =>
  failtest ["In (", inWhat, " ", expString e, "), ", msg]
val checkAssertChecks = checkOneExpChecks "check-assert"
val checkErrorChecks  = checkOneExpChecks "check-error"
```

P.4 Evaluation

The chunks of the evaluator are laid out as follows:

S396e. ⟨*evaluation, testing, and the read-eval-print loop for Typed Impcore* S396e⟩≡ (S391a)

⟨*definitions of* `eval` *and* `evaldef` *for Typed Impcore* S397a⟩
⟨*definitions of* `basis` *and* `processDef` *for Typed Impcore* S392a⟩
⟨*shared definition of* `withHandlers` S239a⟩
⟨*shared unit-testing utilities* S225a⟩
⟨*definition of* `testIsGood` *for Typed Impcore* S395a⟩
⟨*shared definition of* `processTests` S226⟩
⟨*shared read-eval-print loop* S237⟩

In the evaluator, values of unit type need a representation. And all values of unit type must test equal with =, so they must have the *same* representation. Because that representation is the result of evaluating a WHILE loop or an empty BEGIN, it is defined here.

S397a. ⟨*definitions of* eval *and* evaldef *for Typed Impcore* S397a⟩≡ (S396e) S397b ▷

```
val unitVal = NUM 1983
```

ev : exp -> value

The implementation of the evaluator uses the same techniques I use to implement μScheme in Chapter 5. Because of Typed Impcore's many environments, Typed Impcore's evaluator does more bookkeeping.

S397b. ⟨*definitions of* eval *and* evaldef *for Typed Impcore* S397a⟩+≡ (S396e) ◁ S397a S398b ▷

eval : exp * value ref env * func env * value ref env -> value

```
fun projectBool (NUM 0) = false
  | projectBool _       = true

fun eval (e, globals, functions, formals) =
  let val toBool = projectBool
      fun ofBool true    = NUM 1
        | ofBool false   = NUM 0
      fun eq (NUM n1,   NUM n2)   = (n1 = n2)
        | eq (ARRAY a1, ARRAY a2) = (a1 = a2)
        | eq _                    = false
      fun findVar v = find (v, formals)
                      handle NotFound _ => find (v, globals)
      fun ev (LITERAL n)          = n
        | ev (VAR x)              = !(findVar x)
        | ev (SET (x, e))         = let val v = ev e
                                    in  v before findVar x := v
                                    end
        | ev (IFX (cond, t, f))   = if toBool (ev cond) then ev t else ev f
        | ev (WHILEX (cond, exp)) =
            if toBool (ev cond) then
                (ev exp; ev (WHILEX (cond, exp)))
            else
                unitVal
        | ev (BEGIN es) =
            let fun b (e::es, lastval) = b (es, ev e)
                  | b (   [], lastval) = lastval
            in  b (es, unitVal)
            end
        | ev (EQ (e1, e2)) = ofBool (eq (ev e1, ev e2))
        | ev (PRINTLN e)   = (print (valueString (ev e)^"\n"); unitVal)
        | ev (PRINT   e)   = (print (valueString (ev e));      unitVal)
        | ev (APPLY (f, args)) =
            (case find (f, functions)
               of PRIMITIVE p  => p (map ev args)
                | USERDEF func => ⟨apply user-defined function func to args S398a⟩)
        ⟨more alternatives for ev for Typed Impcore 345b⟩
  in  ev e
  end
```

To apply a function, the evaluator builds an environment in which the function's body can be evaluated.. It strips the types off the formals and it puts the actuals in mutable reference cells. The number of actuals should be the same as the number of formals, or the call would have been rejected by the type checker. If the number isn't the same, the evaluator catches exception `BindListLength` and raises `BugInTypeChecking`.

S398a. ⟨*apply user-defined function* `func` *to* `args` S398a⟩≡ (S397b)

```
formals : name      list
actuals : value ref list
```

```
let val (formals, body) = func
    val actuals         = map (ref o ev) args
in  eval (body, globals, functions, mkEnv (formals, actuals))
    handle BindListLength =>
        raise BugInTypeChecking "Wrong number of arguments to function"
end
```

Evaluating a definition produces two environments, plus a string representing the thing defined.

S398b. ⟨*definitions of* `eval` *and* `evaldef` *for Typed Impcore* S397a⟩+≡ (S396e) ◁S397b

```
evaldef : def * value ref env * func env -> value ref env * func env * string
```

```
fun evaldef (d, globals, functions) =
  case d
    of VAL (x, e) => ⟨evaluate e and bind the result to x S398c⟩
     | EXP e      => evaldef (VAL ("it", e), globals, functions)
     | DEFINE (f, { body = e, formals = xs, returns = rt }) =>
         (globals, bind (f, USERDEF (map #1 xs, e), functions), f)
```

S398c. ⟨*evaluate* `e` *and bind the result to* `x` S398c⟩≡ (S398b)

```
let val v = eval (e, globals, functions, emptyEnv)
in  (bind (x, ref v, globals), functions, valueString v)
end
```

P.5 String conversions

To enable the interpreter to issue error messages, I define a string-conversion function for values, plus one for each major syntactic category.

To distinguish an array from a list, the elements of an array are shown in *square* brackets.

S398d. ⟨*definition of* `valueString` *for Typed Impcore* S398d⟩≡ (S391b)

```
fun valueString (NUM n) = intString n
  | valueString (ARRAY a) =
      if Array.length a = 0 then
          "[]"
      else
          let val eltStrings = map valueString (Array.foldr op :: [] a)
          in  String.concat ["[", spaceSep eltStrings, "]"]
          end
```

This code prints types.

S398e. ⟨*definitions of* `typeString` *and* `funtyString` *for Typed Impcore* S398e⟩≡ S398f▷

```
fun typeString BOOLTY        = "bool"
  | typeString INTTY         = "int"
  | typeString UNITTY        = "unit"
  | typeString (ARRAYTY tau) = "(array " ^ typeString tau ^ ")"
```

S398f. ⟨*definitions of* `typeString` *and* `funtyString` *for Typed Impcore* S398e⟩+≡ ◁S398e

```
fun funtyString (FUNTY (args, result)) =
  "(" ^ spaceSep (map typeString args) ^ " -> " ^ typeString result ^ ")"
```

And expressions.

S399a. ⟨*definition of* `expString` *for Typed Impcore* S399a⟩≡ (S391b)

```
fun expString e =
  let fun bracket s = "(" ^ s ^ ")"
      val bracketSpace = bracket o spaceSep
      fun exps es = map expString es
  in  case e
        of LITERAL v => valueString v
         | VAR name => name
         | SET (x, e) => bracketSpace ["set", x, expString e]
         | IFX (e1, e2, e3) => bracketSpace ("if" :: exps [e1, e2, e3])
         | WHILEX (cond, body) =>
             bracketSpace ["while", expString cond, expString body]
         | BEGIN es => bracketSpace ("begin" :: exps es)
         | EQ (e1, e2) => bracketSpace ("=" :: exps [e1, e2])
         | PRINTLN e => bracketSpace ["println", expString e]
         | PRINT e => bracketSpace ["print", expString e]
         | APPLY (f, es) => bracketSpace (f :: exps es)
         | AAT (a, i) => bracketSpace ("array-at" :: exps [a, i])
         | APUT (a, i, e) => bracketSpace ("array-put" :: exps [a, i, e])
         | AMAKE (e, n) => bracketSpace ("make-array" :: exps [e, n])
         | ASIZE a => bracketSpace ("array-size" :: exps [a])
  end
```

And definitions.

S399b. ⟨*definitions of* `defString` *and* `defName` *for Typed Impcore* S399b⟩≡ (S391b)

```
fun defString d =
  let fun bracket s = "(" ^ s ^ ")"
      val bracketSpace = bracket o spaceSep
      fun formal (x, t) = "[" ^ x ^ " : " ^ typeString t ^ "]"
  in  case d
        of EXP e => expString e
         | VAL (x, e) => bracketSpace ["val", x, expString e]
         | DEFINE (f, { formals, body, returns }) =>
             bracketSpace ["define", typeString returns, f,
                                  bracketSpace (map formal formals), expString body]
  end
fun defName (VAL (x, _)) = x
  | defName (DEFINE (x, _)) = x
  | defName (EXP _) =
      raise InternalError "asked for name defined by expression"
```

P.6 Parsing

Typed Impcore reuses almost all of μScheme's lexical analysis.

S399c. ⟨*lexical analysis and parsing for Typed Impcore, providing* `filexdefs` *and* `stringsxdefs` S399c⟩

⟨*lexical analysis for* μ*Scheme and related languages* S383d⟩
⟨*parsers for single tokens for* μ*Scheme-like languages* S385a⟩
⟨*parsers for single tokens for Typed Impcore* S400a⟩
⟨*parsers and parser builders for formal parameters and bindings* S385c⟩
⟨*parser builders for typed languages* S401b⟩
⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩
⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S233a⟩

Parsers for single tokens

Typed Impcore does need its own list of reserved words.

S400a. ⟨*parsers for single tokens for Typed Impcore* S400a⟩≡ (S399c) S400b ▷

```
val reserved = [ "if", "while", "set", "begin", "println", "print", "="
                 , "array-at", "array-put", "make-array", "array-size"
                 , "val", "define", "use"
                 , "check-expect", "check-assert", "check-error"
                 , "check-type-error", "check-function-type"
                 ]
val name = rejectReserved reserved <$>! namelike
```

To allow relatively lightweight concrete syntax for function types, an arrow does not parse as a name.

S400b. ⟨*parsers for single tokens for Typed Impcore* S400a⟩+≡ (S399c) ◁ S400a

```
val name  = sat (fn n => n <> "->") name
val arrow = (fn (NAME "->") => SOME () | _ => NONE) <$>? pretoken
```

Parsers for expressions

S400c. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩≡ (S399c) S400d ▷

```
exp      : exp parser
exptable : exp parser -> exp parser
```

```
fun exptable exp = usageParsers
  [ ("(if e1 e2 e3)",      curry3 IFX     <$> exp <*> exp <*> exp)
  , ("(while e1 e2)",      curry  WHILEX  <$> exp <*> exp)
  , ("(set x e)",          curry  SET     <$> name <*> exp)
  , ("(begin e ...)",             BEGIN   <$> many exp)
  , ("(println e)",               PRINTLN <$> exp)
  , ("(print e)",                 PRINT   <$> exp)
  , ("(= e1 e2)",          curry  EQ      <$> exp <*> exp)
  , ("(array-at a i)",     curry  AAT     <$> exp <*> exp)
  , ("(array-put a i e)",  curry3 APUT    <$> exp <*> exp <*> exp)
  , ("(make-array n e)",   curry  AMAKE   <$> exp <*> exp)
  , ("(array-size a)",            ASIZE   <$> exp)
  ]
```

S400d. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩+≡ (S399c) ◁ S400c S400e ▷

```
val atomicExp =  VAR     <$> name
             <|> LITERAL <$> NUM <$> int
             <|> booltok <!> "Typed Impcore has no Boolean literals"
             <|> quote   <!> "Typed Impcore has no quoted literals"
```

S400e. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩+≡ (S399c) ◁ S400d S400f ▷

```
fun impcorefun what exp =  name
                       <|> exp <!> ("only named functions can be " ^ what)
                       <?> "function name"
```

S400f. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩+≡ (S399c) ◁ S400e S401a ▷

```
fun exp tokens = (
     atomicExp
 <|> exptable exp
 <|> leftCurly <!> "curly brackets are not supported"
 <|> left *> right <!> "empty application"
 <|> bracket("function application",
             curry APPLY <$> impcorefun "applied" exp <*> many exp)
) tokens
```

Parsers for types and definitions

Typed Impcore has only the three base types, plus whatever types are formed using `array`.

S401a. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩+≡ (S399c) ◁S400f S401c▷

```
ty    : ty    parser
funty : funty parser
```

```
fun badTypeName (loc, n) =
      synerrorAt ("Cannot recognize name " ^ n ^ " as a type") loc
fun repeatable_ty tokens = (
      BOOLTY <$ kw "bool"
  <|> UNITTY <$ kw "unit"
  <|> INTTY  <$ kw "int"
  <|> badTypeName <$>! @@ name
  <|> usageParsers [("(array ty)", ARRAYTY <$> ty)]
  ) tokens
and ty tokens = (repeatable_ty <?> "int, bool, unit, or (array ty)") tokens

val funty = bracket ("function type",
                     curry FUNTY <$> many repeatable_ty <* arrow <*> ty)
```

A function definition has a list of typed formal parameters. Each formal parameter is bound to a type, and the names must be mutually distinct.

S401b. ⟨*parser builders for typed languages* S401b⟩≡ (S399c S413b)

```
typedFormalsOf : string parser -> 'b parser -> 'a parser
                                  -> string -> (string * 'a) list parser
```

```
fun typedFormalOf name colon ty =
      bracket ("[x : ty]", pair <$> name <* colon <*> ty)
fun typedFormalsOf name colon ty context =
  let val formal = typedFormalOf name colon ty
  in  distinctBsIn (bracket("(... [x : ty] ...)", many formal)) context
  end
```

The `deftable` parser parses the true definitions.

S401c. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩+≡ (S399c) ◁S401a S401d▷

```
fun define ty f formals body =
  DEFINE (f, { returns = ty, formals = formals, body = body })
val formals =
      typedFormalsOf name (kw ":") ty "formal parameters in 'define'"
val deftable = usageParsers
  [ ("(define ty f (args) body)",
         define <$> ty <*> name <*> formals <*> exp)
  , ("(val x e)",
         curry VAL <$> name <*> exp)
  ]
```

Function `unit_test` parses the unit tests.

S401d. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩+≡ (S399c) ◁S401c S402a▷

```
testtable : unit_test parser
```

```
val testtable = usageParsers
  [ ("(check-expect e1 e2)", curry CHECK_EXPECT <$> exp <*> exp)
  , ("(check-assert e)",           CHECK_ASSERT <$> exp)
  , ("(check-error e)",            CHECK_ERROR  <$> exp)
  , ("(check-type-error d)",
                 CHECK_TYPE_ERROR    <$> (deftable <|> EXP <$> exp))
  , ("(check-function-type f (tau ... -> tau))",
          curry CHECK_FUNCTION_TYPE <$> impcorefun "checked" exp <*> funty)
  ]
```

And `xdef` parses all the extended definitions.

S402a. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩+≡ (S399c) ◁S401d S402b▷

```
val xdeftable = usageParsers
  [ ("(use filename)", USE <$> name)
  ⟨rows added to Typed Impcore xdeftable in exercises S402c⟩
  ]

val xdef =  DEF  <$> deftable
        <|> TEST <$> testtable
        <|>          xdeftable
        <|> badRight "unexpected right bracket"
        <|> DEF <$> EXP <$> exp
        <?> "definition"
```

S402b. ⟨*parsers and* `xdef` *streams for Typed Impcore* S400c⟩+≡ (S399c) ◁S402a

```
val xdefstream = interactiveParsedStream (schemeToken, xdef)
```

S402c. ⟨*rows added to Typed Impcore* `xdeftable` *in exercises* S402c⟩≡ (S402a)

```
(* add syntactic extensions here, each preceded by a comma *)
```

Appendix Q contents

Supporting code for the Typed μScheme interpreter Q

Q.1 Organizing code chunks into an interpreter

The overall structure of the Typed μScheme interpreter is similar to the structure of the Typed Impcore interpreter. But in addition to evaluation and type checking, it also has kind checking.

S405a. ⟨*tuscheme.sml* S405a⟩≡
  ⟨*exceptions used in languages with type checking* S213c⟩
  ⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S213a⟩

  ⟨*kinds for typed languages* 355a⟩
  ⟨*types for Typed μScheme* S410a⟩
  ⟨*sets of free type variables in Typed μScheme* 371⟩
  ⟨*shared utility functions on sets of type variables* (from chunk 697b)⟩
  ⟨*kind checking for Typed μScheme* 378b⟩

  ⟨*abstract syntax and values for Typed μScheme* S405b⟩
  ⟨*utility functions on values (μScheme, Typed μScheme, nano-ML)* 308a⟩

  ⟨*capture-avoiding substitution for Typed μScheme* 374⟩
  ⟨*type equivalence for Typed μScheme* 370a⟩
  ⟨*type checking for Typed μScheme* (from chunk 697b)⟩

  ⟨*lexical analysis and parsing for Typed μScheme, providing* `filexdefs` *and* `stringsxdefs` S413b⟩

  ⟨*evaluation, testing, and the read-eval-print loop for Typed μScheme* S406b⟩

  ⟨*implementations of Typed μScheme primitives and definition of* `initialBasis` S406d⟩
  ⟨*function* `runStream`*, which evaluates input given* `initialBasis` S240b⟩
  ⟨*look at command-line arguments, then run* S240c⟩

The definition of `xdef` is, as usual, shared, and less usually, the definition of `valueString` is shared with μScheme and nano-ML.

S405b. ⟨*abstract syntax and values for Typed μScheme* S405b⟩≡ (S405a)
  ⟨*definitions of* `exp` *and* `value` *for Typed μScheme* 361a⟩
  ⟨*definition of* `def` *for Typed μScheme* 361c⟩
  ⟨*definition of* `unit_test` *for explicitly typed languages* S406a⟩
  ⟨*definition of* `xdef` *(shared)* S214b⟩
  ⟨*definition of* `valueString` *for μScheme, Typed μScheme, and nano-ML* 307b⟩
  ⟨*definition of* `expString` *for Typed μScheme* S410b⟩
  ⟨*definitions of* `defString` *and* `defName` *for Typed μScheme* S410c⟩

The new unit-test forms `check-type` and `check-type-error`, which are meaningful only in typed languages, demand new abstract syntax.

S406a. ⟨*definition of* `unit_test` *for explicitly typed languages* S406a⟩≡ (S405b)

```
datatype unit_test = CHECK_EXPECT      of exp * exp
                   | CHECK_ASSERT      of exp
                   | CHECK_ERROR       of exp
                   | CHECK_TYPE        of exp * tyex
                   | CHECK_TYPE_ERROR  of def
```

The core of the interpreter mixes reusable chunks with chunks used only in Typed μScheme.

S406b. ⟨*evaluation, testing, and the read-eval-print loop for Typed μScheme* S406b⟩≡ (S405a)

```
⟨definition of namedValueString for functional bridge languages S413a⟩
⟨definitions of eval and evaldef for Typed μScheme S411b⟩
⟨definitions of basis and processDef for Typed μScheme S406c⟩
⟨shared definition of withHandlers S239a⟩
⟨shared unit-testing utilities S225a⟩
⟨definition of testIsGood for Typed μScheme S409a⟩
⟨shared definition of processTests S226⟩
⟨shared read-eval-print loop S237⟩
```

Function `processDef` is very similar to the `processDef` used in Typed Impcore, but it accounts for the different environments used in the basis. Instead of two environments to store the types of variables and one to store the types of functions, Typed μScheme has just the one environment Γ (`Gamma`), which stores the type of each identifier, whether it is a variable or a function. And Typed μScheme's basis also includes a kind environment Δ (`Delta`), which stores the kind of each type constructor.

S406c. ⟨*definitions of* `basis` *and* `processDef` *for Typed μScheme* S406c⟩≡ (S406b)

```
processDef : def * basis * interactivity -> basis
```

```
⟨definition of basis for Typed μScheme 382b⟩
fun processDef (d, (Delta, Gamma, rho), interactivity) =
  let val (Gamma, tystring)  = typdef (d, Delta, Gamma)
      val (rho,   valstring) = evaldef (d, rho)
      val _ = if echoes interactivity then
                println (valstring ^ " : " ^ tystring)
              else
                ()
  in  (Delta, Gamma, rho)
  end
```

Q.2 Primitive functions, predefined functions, and the initial basis

The initial basis is built from primitives and from predefined functions. The ⟨*functions for building primitives when types are checked* S393a⟩ (from Appendix P) are reused.

S406d. ⟨*implementations of Typed μScheme primitives and definition of* `initialBasis` S406d⟩≡ (S405a)

```
⟨functions for building primitives when types are checked S393a⟩
⟨utility functions and types for making Typed μScheme primitives S407a⟩
⟨definition of primBasis for Typed μScheme 382c⟩
val primitiveBasis = primBasis
val predefs    = ⟨predefined Typed μScheme functions, as strings (from chunk S407e)⟩
val initialBasis =
  let val xdefs = stringsxdefs ("predefined functions", predefs)
  in  readEvalPrintWith predefinedFunctionError (xdefs, primBasis, noninteractive)
  end
```

Q.2.1 Primitive functions of Typed *μ*Scheme

The primitives resemble the primitives in Chapter 5, except that each primitive comes with a type as well as a value.

A comparison takes two arguments. Most comparisons (but not equality) apply only to integers.

S407a. ⟨*utility functions and types for making Typed μScheme primitives* S407a⟩≡ (S406d)

```
comparison : (value * value -> bool) -> (value list -> value)
intcompare : (int   * int   -> bool) -> (value list -> value)
comptype   : tyex
```

```
fun comparison f = binaryOp (BOOLV o f)
fun intcompare f =
      comparison (fn (NUM n1, NUM n2) => f (n1, n2)
                   | _ => raise BugInTypeChecking "comparing non-numbers")
val comptype = FUNTY ([inttype, inttype], booltype)
```

S407b. ⟨*primitive functions for Typed μScheme* :: S407b⟩≡ (382c) S407c ▷

```
("<", intcompare op <, comptype) ::
(">", intcompare op >, comptype) ::
("=", comparison equalatoms,
                        FORALL (["'a"], FUNTY ([tvA, tvA], booltype))) ::
```

Two of the print primitives have polymorphic types.

S407c. ⟨*primitive functions for Typed μScheme* :: S407b⟩+≡ (382c) ◁ S407b

```
("println", unaryOp (fn x => (print (valueString x^"\n"); unitVal)),
   FORALL (["'a"], FUNTY ([tvA], unittype))) ::
("print", unaryOp (fn x => (print (valueString x);       unitVal)),
   FORALL (["'a"], FUNTY ([tvA], unittype))) ::
("printu",  unaryOp (fn NUM n => (printUTF8 n; unitVal)
                      | v => raise BugInTypeChecking "printu of non-number"),
   FUNTY ([inttype], unittype)) ::
```

In plain Typed *μ*Scheme, all the primitives are functions, so this chunk is empty. But in case you want to define a non-function primitive for one of the exercises in Chapter 6, I leave an (empty) place for primitives that aren't functions.

S407d. ⟨*primitives that aren't functions, for Typed μScheme* :: S407d⟩≡ (382c)

```
(* fill in non-function primitives here *)
```

Q.2.2 Predefined functions of Typed *μ*Scheme

Because programming in Typed *μ*Scheme is a hassle, Typed *μ*Scheme has fewer predefined functions than *μ*Scheme. Some of these functions are defined in Chapter 6. The rest are here.

Becauses lists in Typed *μ*Scheme are homogeneous, the funny list functions built from `car` and `cdr` are much less useful than in *μ*Scheme.

S407e. ⟨*predefined Typed μScheme functions* S407e⟩≡ S408a ▷

```
(val caar
   (type-lambda ('a)
      (lambda ([xs : (list (list 'a))])
         ((@ car 'a) ((@ car (list 'a)) xs)))))
(val cadr
   (type-lambda ('a)
      (lambda ([xs : (list (list 'a))])
         ((@ car (list 'a)) ((@ cdr (list 'a)) xs)))))
```

The Boolean functions are defined almost exactly as in Typed Impcore.

S408a. ⟨*predefined Typed μScheme functions* S407e⟩+≡ ◁ S407e S408b ▷

```
(define bool and ([b : bool] [c : bool]) (if b  c  b))
(define bool or  ([b : bool] [c : bool]) (if b  b  c))
(define bool not ([b : bool])            (if b #f #t))
```

Integer comparisons are defined as in Typed Impcore, but to define `!=` requires a type abstraction. This is progress! In Typed Impcore, a polymorphic `!=` couldn't be defined as as function; it would have to be a syntactic form.

S408b. ⟨*predefined Typed μScheme functions* S407e⟩+≡ ◁ S408a S408c ▷

```
(define bool <= ([x : int] [y : int]) (not (> x y)))
(define bool >= ([x : int] [y : int]) (not (< x y)))
(val != (type-lambda ('a) (lambda ([x : 'a] [y : 'a]) (not ((@ = 'a) x y)))))
```

Integer functions are almost as in Typed Impcore. The only difference is that in Typed μScheme, equality is a primitive, polymorphic function, and it must be instantiated before use.

S408c. ⟨*predefined Typed μScheme functions* S407e⟩+≡ ◁ S408b S408d ▷

```
(define int max ([m : int] [n : int]) (if (> m n) m n))
(define int min ([m : int] [n : int]) (if (< m n) m n))
(define int mod ([m : int] [n : int]) (- m (* n (/ m n))))
(define int gcd ([m : int] [n : int]) (if ((@ = int) n 0) m (gcd n (mod m n))))
(define int lcm ([m : int] [n : int]) (* m (/ n (gcd m n))))
```

Typed μScheme provides a subset of the list functions found in μScheme. Most of them appear in Chapter 6, and some (`exists?`, `all?`, and `foldr`) are left as exercises. The remaining predefined function is `append`, which is defined using the model shown in Chapter 6: a function that is both polymorphic and recursive is written using a `letrec` inside a `type-lambda`.

S408d. ⟨*predefined Typed μScheme functions* S407e⟩+≡ ◁ S408c

```
(val append
  (type-lambda ('a)
     (letrec [([append-mono : ((list 'a) (list 'a) -> (list 'a))]
                 (lambda ([xs : (list 'a)] [ys : (list 'a)])
                   (if ((@ null? 'a) xs)
                      ys
                      ((@ cons 'a)
                          ((@ car 'a) xs)
                          (append-mono ((@ cdr 'a) xs) ys)))))]
        append-mono)))
```

Q.3 Unit testing

As in untyped μScheme, equality for `check-expect` is structural.

S408e. ⟨*utility functions on μScheme, Typed μScheme, and nano-ML values* **[[tuscheme]]** S408e⟩≡

```
fun testEquals (PAIR (car1, cdr1), PAIR (car2, cdr2)) =
      testEquals (car1, car2) andalso testEquals (cdr1, cdr2)
  | testEquals (v1, v2) = equalatoms (v1, v2)
```

The testIsGood function for Typed μScheme mirrors the testIsGood function for Typed Impcore, but the environments are different. Because the tests are also different, I didn't try to share the ty or outcome functions.

S409a. ⟨*definition of* testIsGood *for Typed μScheme* S409a⟩≡ (S406b)

```
fun testIsGood (test, (Delta, Gamma, rho)) =
  let fun ty e = typeof (e, Delta, Gamma)
                 handle NotFound x =>
                   raise TypeError ("name " ^ x ^ " is not defined")
      ⟨shared check{Expect,Assert,Error,Type{Checks, which call ty S409b⟩
      fun checks (CHECK_EXPECT (e1, e2)) = checkExpectChecks (e1, e2)
        | checks (CHECK_ASSERT e)        = checkAssertChecks e
        | checks (CHECK_ERROR e)         = checkErrorChecks e
        | checks (CHECK_TYPE (e, tau))   = checkTypeChecks (e, tau)
        | checks (CHECK_TYPE_ERROR e)    = true

      fun outcome e =
        withHandlers (fn () => OK (eval (e, rho))) () (ERROR o stripAtLoc)
      ⟨asSyntacticValue for μScheme, Typed Impcore, Typed μScheme, and nano-ML S383a⟩
      ⟨shared check{Expect,Assert,Error{Passes, which call outcome S224d⟩
      fun deftystring d =
        snd (typdef (d, Delta, Gamma))
        handle NotFound x =>
          raise TypeError ("name " ^ x ^ " is not defined")
      ⟨shared checkTypePasses and checkTypeErrorPasses, which call ty S396a⟩
      fun passes (CHECK_EXPECT (c, e)) = checkExpectPasses (c, e)
        | passes (CHECK_ASSERT c)      = checkAssertPasses c
        | passes (CHECK_ERROR c)       = checkErrorPasses  c
        | passes (CHECK_TYPE (c, tau)) = checkTypePasses   (c, tau)
        | passes (CHECK_TYPE_ERROR d)  = checkTypeErrorPasses d

  in  checks test andalso passes test
  end
```

Function checks confirms that forms check-expect, check-error, and check-type contain only expressions that typecheck. But not check-type-error. The whole point of check-type-error is that its expression *doesn't* typecheck. Thus, it is not typechecked by function check. Instead, it is typechecked by function passes—if it has a type, it fails.

Most of the checking functions are implemented in Appendices O and P. But checkTypeChecks is implemented here.

S409b. ⟨*shared* check{Expect,Assert,Error,Type{Checks*, which call* ty S409b⟩≡ (S409a S395a)

```
fun checkTypeChecks (e, tau) =
  let val tau' = ty e
  in  true
  end
  handle TypeError msg =>
    failtest ["In (check-type ", expString e, " " ^ typeString tau, "), ", msg]
```

Q.4 String conversions

Types, expressions, and definitions can all be rendered as strings.

S410a. ⟨*types for Typed μScheme* S410a⟩≡ (S405a)

```
fun typeString (TYCON c) = c
  | typeString (TYVAR a) = a
  | typeString (FUNTY (args, result)) =
      "(" ^ spaceSep (map typeString args) ^ " -> " ^ typeString result ^ ")"
  | typeString (CONAPP (tau, [])) = "(" ^ typeString tau ^ ")"
  | typeString (CONAPP (tau, tys)) =
      "(" ^ typeString tau ^ " " ^ spaceSep (map typeString tys) ^ ")"
  | typeString (FORALL (tyvars, tau)) =
      "(forall [" ^ spaceSep tyvars ^ "] " ^ typeString tau ^ ")"
```

S410b. ⟨*definition of* `expString` *for Typed μScheme* S410b⟩≡ (S405b)

```
fun expString e =
  let fun bracket s = "(" ^ s ^ ")"
      val bracketSpace = bracket o spaceSep
      fun exps es = map expString es
      fun formal (x, tau) = bracketSpace [typeString tau, x]
      fun withBindings (keyword, bs, e) =
        bracket (spaceSep [keyword, bindings bs, expString e])
      and bindings bs = bracket (spaceSep (map binding bs))
      and binding (x, e) = bracket (x ^ " " ^ expString e)

      fun tybinding ((x, ty), e) = bracketSpace [formal (x, ty), expString e]
      and tybindings bs = bracket (spaceSep (map tybinding bs))
      val letkind = fn LET => "let" | LETSTAR => "let*"
  in  case e
        of LITERAL v         => valueString v
         | VAR name          => name
         | SET (x, e)        => bracketSpace ["set", x, expString e]
         | IFX (e1, e2, e3)  => bracketSpace ("if" :: exps [e1, e2, e3])
         | WHILEX (cond, body) =>
                      bracketSpace ["while", expString cond, expString body]
         | BEGIN es          => bracketSpace ("begin" :: exps es)
         | APPLY (e, es)     => bracketSpace (exps (e::es))
         | LETX (lk, bs, e)  => bracketSpace [letkind lk, bindings bs, expString e]
         | LETRECX (bs, e)  => bracketSpace ["letrec", tybindings bs, expString e]
         | LAMBDA (xs, e)    =>
             bracketSpace ["lambda", bracketSpace (map formal xs), expString e]
         | TYLAMBDA (alphas, e) =>
             bracketSpace ["type-lambda", bracketSpace alphas, expString e]
         | TYAPPLY (e, taus) =>
             bracketSpace ("@" :: expString e :: map typeString taus)
  end
```

S410c. ⟨*definitions of* `defString` *and* `defName` *for Typed μScheme* S410c⟩≡ (S405b) S411a ▷

```
fun defString d =
  let fun bracket s = "(" ^ s ^ ")"
      val bracketSpace = bracket o spaceSep
      fun formal (x, t) = "[" ^ x ^ " : " ^ typeString t ^ "]"
  in  case d
        of EXP e => expString e
         | VAL (x, e) => bracketSpace ["val", x, expString e]
         | VALREC (x, tau, e) =>
             bracketSpace ["val-rec", formal (x, tau), expString e]
         | DEFINE (f, rtau, (formals, body)) =>
             bracketSpace ["define", typeString rtau, f,
                           bracketSpace (map formal formals), expString body]
  end
```

S411a. ⟨*definitions of* `defString` *and* `defName` *for Typed μScheme* S410c⟩+≡ (S405b) ◁S410c

```
fun defName (VAL (x, _))        = x
  | defName (VALREC (x, _, _)) = x
  | defName (DEFINE (x, _, _)) = x
  | defName (EXP _) = raise InternalError "asked for name defined by expression"
```

Q.5 Evaluation

The implementation of the evaluator is almost identical to the implementation in Chapter 5. There are only two significant differences:

- The abstract syntax `LAMBDA` has types, but the value `CLOSURE` does not.
- The evaluator needs cases for `TYAPPLY` and `TYLAMBDA`.

Another difference is that many potential run-time errors should be impossible because the relevant code would be rejected by the type checker. If one of those errors occurs anyway, the evaluator raises the exception `BugInTypeChecking`, not `RuntimeError`.

S411b. ⟨*definitions of* `eval` *and* `evaldef` *for Typed μScheme* S411b⟩≡ (S406b) S412d▷

```
eval : exp * value ref env -> value
ev   : exp                 -> value
```

```
fun eval (e, rho) =
  let fun ev (LITERAL n) = n
        ⟨alternatives for ev for TYAPPLY and TYLAMBDA 380b⟩
        ⟨more alternatives for ev for Typed μScheme S411c⟩
  in  ev e
  end
```

Code for variables is just as in Chapter 5.

S411c. ⟨*more alternatives for* `ev` *for Typed μScheme* S411c⟩≡ (S411b) S411d▷

```
| ev (VAR v) = !(find (v, rho))
| ev (SET (n, e)) =
    let val v = ev e
    in  find (n, rho) := v;
        v
    end
```

Code for control flow is just as in Chapter 5.

S411d. ⟨*more alternatives for* `ev` *for Typed μScheme* S411c⟩+≡ (S411b) ◁S411c S411e▷

```
| ev (IFX (e1, e2, e3)) = ev (if projectBool (ev e1) then e2 else e3)
| ev (WHILEX (guard, body)) =
    if projectBool (ev guard) then
      (ev body; ev (WHILEX (guard, body)))
    else
      unitVal
| ev (BEGIN es) =
    let fun b (e::es, lastval) = b (es, ev e)
          | b (   [], lastval) = lastval
    in  b (es, unitVal)
    end
```

Code for a `lambda` removes the types from the abstract syntax.

S411e. ⟨*more alternatives for* `ev` *for Typed μScheme* S411c⟩+≡ (S411b) ◁S411d S412a▷

```
| ev (LAMBDA (args, body)) = CLOSURE ((map (fn (x, ty) => x) args, body), rho)
```

Code for application is almost as in Chapter 5, except if the program tries to apply a non-function, the evaluator raises `BugInTypeChecking`, not `RuntimeError`, because the type checker should reject any program that could apply a non-function.

S412a. ⟨*more alternatives for* `ev` *for Typed μScheme* S411c⟩+≡ (S411b) ◁S411e S412b▷

```
  | ev (APPLY (f, args))  =
         (case ev f
            of PRIMITIVE prim => prim (map ev args)
             | CLOSURE clo => ⟨apply closure clo to args 310c⟩
             | v => raise BugInTypeChecking "applied non-function"
         )
```

Code for the `LETX` family is as in Chapter 5.

S412b. ⟨*more alternatives for* `ev` *for Typed μScheme* S411c⟩+≡ (S411b) ◁S412a S412c▷

```
  | ev (LETX (LET, bs, body)) =
      let val (names, values) = ListPair.unzip bs
      in  eval (body, rho <+> mkEnv (names, map (ref o ev) values))
      end
  | ev (LETX (LETSTAR, bs, body)) =
      let fun step ((n, e), rho) = bind (n, ref (eval (e, rho)), rho)
      in  eval (body, foldl step rho bs)
      end
```

S412c. ⟨*more alternatives for* `ev` *for Typed μScheme* S411c⟩+≡ (S411b) ◁S412b

```
  | ev (LETRECX (bs, body)) =
      let val (tynames, values) = ListPair.unzip bs
          val names = map fst tynames
          val rho' = rho <+> mkEnv (names, map (fn _ => ref (unspecified())) values)
          val updates = map (fn ((x, _), e) => (x, eval (e, rho'))) bs
      in  List.app (fn (x, v) => find (x, rho') := v) updates;
          eval (body, rho')
      end
```

Evaluating a definition can produce a new environment. The function `evaldef` also returns a string which, if nonempty, should be printed to show the value of the item. Type soundness requires a change in the evaluation rule for `VAL`; as described in Exercise 46 in Chapter 2, `VAL` must always create a new binding.

S412d. ⟨*definitions of* `eval` *and* `evaldef` *for Typed μScheme* S411b⟩+≡ (S406b) ◁S411b

`evaldef : def * value ref env -> value ref env * string`

```
  fun evaldef (VAL (x, e), rho) =
        let val v   = eval (e, rho)
            val rho = bind (x, ref v, rho)
        in  (rho, namedValueString x v)
        end
    | evaldef (VALREC (x, tau, e), rho) =
        let val this = ref NIL
            val rho' = bind (x, this, rho)
            val v    = eval (e, rho')
            val _    = this := v
        in  (rho', namedValueString x v)
        end
    | evaldef (EXP e, rho) = (* differs from VAL ("it", e) only in its response *)
        let val v   = eval (e, rho)
            val rho = bind ("it", ref v, rho)
        in  (rho, valueString v)
        end
    | evaldef (DEFINE (f, tau, lambda), rho) =
        evaldef (VALREC (f, tau, LAMBDA lambda), rho)
```

In the `VALREC` case, the interpreter evaluates `e` while name `x` is still bound to a location that contains `NIL`—that is, before the assignment to `this`. Therefore, as described on page 362, evaluating `e` must not evaluate `x`—because the mutable cell for `x` does not yet contain its correct value. Evaluation is prevented in the parser, which issues a syntax error unless `e` is a `LAMBDA`.

Function `evaldef` returns a string identifying what was just defined. If the thing just defined is a function, `evaldef` identifies it by its name. Otherwise, it shows the actual value.

S413a. ⟨*definition of* `namedValueString` *for functional bridge languages* S413a⟩≡ (S406b)

`namedValueString : name -> value -> string`

```
fun namedValueString x v =
  case v of CLOSURE _ => x
          | PRIMITIVE _ => x
          | _ => valueString v
```

Q.6 Lexical analysis and parsing

S413b. ⟨*lexical analysis and parsing for Typed μScheme, providing* `filexdefs` *and* `stringsxdefs` S413b
⟨*lexical analysis for μScheme and related languages* S383d⟩
⟨*parsers for single tokens for μScheme-like languages* S385a⟩
⟨*parsers for Typed μScheme tokens* S413c⟩
⟨*parsers and parser builders for formal parameters and bindings* S385c⟩
⟨*parsers and parser builders for Scheme-like syntax* S386d⟩
⟨*parser builders for typed languages* S414c⟩
⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩
⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S233a⟩

Typed μScheme reuses most of μScheme's lexical analysis, but to simplify the parsing of function types, the arrow `->` is treated as a reserved word, not as a name.

To help with error messages, the parser tracks which keywords are used to write expressions and which are used to write types.

S413c. ⟨*parsers for Typed μScheme tokens* S413c⟩≡ (S413b) S414a ▷

```
val expKeywords = [ "if", "while", "set", "begin", "lambda"
                  , "type-lambda", "let", "let*", "letrec", "quote", "@"
                  ]
val tyKeywords  = ["forall", "->"]

val defKeywords = [ "val", "define", "use"
                  , "check-expect", "check-assert", "check-error"
                  , "check-type", "check-type-error"
                  ]

val reserved = expKeywords @ tyKeywords @ defKeywords

fun keyword words =
  let fun isKeyword s = List.exists (fn s' => s = s') words
  in  sat isKeyword namelike
  end

val expKeyword = keyword expKeywords
val tyKeyword  = keyword tyKeywords
val name =
  rejectReserved reserved <$>! sat (curry op <> "->") namelike
```

A type variable begins with a quote mark.

S414a. ⟨*parsers for Typed μScheme tokens* S413c⟩+≡ (S413b) ◁S413c

```
tyvar : name parser
```

```
val arrow = (fn (NAME "->") => SOME () | _ => NONE) <$>? pretoken
val tyvar =
  quote *> (curry op ^ "'" <$> name <?> "type variable (got quote mark)")
```

Formal parameters, whether to `lambda` or `type-lambda`, must not have duplicates.

S414b. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩≡ (S413b) S415a ▷

```
val tlformals =
  nodups ("formal type parameter", "type-lambda") <$>! @@ (many name)

fun nodupsty what (loc, xts) =
  nodups what (loc, map fst xts) >>=+ (fn _ => xts)
                (* error on duplicate names *)

fun letDups LETSTAR (_, bindings) = OK bindings
  | letDups LET bindings = nodupsty ("bound variable", "let") bindings
```

Function `distinctTyvars` is used in multiple interpreters.

S414c. ⟨*parser builders for typed languages* S414c⟩≡ (S413b S399c) S414d ▷

```
distinctTyvars : name list parser
```

```
val distinctTyvars =
  nodups ("quantified type variable", "forall") <$>! @@ (many tyvar)
```

Parsing a type in brackets is tricky enough that I define a function just for that job. The `arrows` function takes two lists: the types that appear before the first arrow, and a list of lists of types that appear after an arrow. To make it usable with languages beyond Typed μScheme, I abstract over the `conapp` and `funty` functions that are used to build types.

S414d. ⟨*parser builders for typed languages* S414c⟩+≡ (S413b S399c) ◁S414c S416b ▷

```
arrowsOf : ('ty * 'ty list -> 'ty) -> ('ty list * 'ty -> 'ty)
        -> 'ty list -> 'ty list list -> 'ty error
```

```
fun arrowsOf conapp funty =
  let fun arrows []              [] = ERROR "empty type ()"
        | arrows (tycon::tyargs) [] = OK (conapp (tycon, tyargs))
        | arrows args            [rhs] =
            (case rhs
               of [result] => OK (funty (args, result))
                | [] => ERROR "no result type after function arrow"
                | _  => ERROR "multiple result types after function arrow")
        | arrows args (_::_::_) = ERROR "multiple arrows in function type"
  in  fn xs => errorLabel "syntax error: " o arrows xs
  end
```

The type parser rejects any keyword that is normally used to write expressions.

S415a. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩+≡ (S413b) ◁S414b S415b▷

```
ty : tyex parser
```

```
val arrows = arrowsOf CONAPP FUNTY
fun ty tokens =
  let fun badExpKeyword (loc, bad) =
        synerrorAt ("looking for type but found `" ^ bad ^ "'") loc
  in     TYCON <$> name
     <|> TYVAR <$> tyvar
     <|> bracketKeyword (kw "forall", "(forall [tyvars] type)",
                         curry FORALL <$> bracket ("('a ...)", distinctTyvars)
                                      <*> ty)
     <|> badExpKeyword <$>! (left *> @@ expKeyword <* matchingRight)
     <|> bracket ("type application or function type",
                  arrows <$> many ty <*>! many (arrow *> many ty))
     <|> int     <!> "expected type; found integer"
     <|> booltok <!> "expected type; found Boolean literal"
  end tokens
```

The parser for formal parameters detects two common mistakes: forgetting to put a space before the colon that separates name from type, and forgetting to wrap a one-element parameter list in brackets. First the colon.

S415b. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩+≡ (S413b) ◁S415a S415c▷

```
endsColon    : name -> name option
badColonParm : (name * tyex) parser
```

```
fun endsColon s =
  if size s > 1 andalso String.sub (s, size s - 1) = #":" then
    SOME (String.substring (s, 0, size s - 1))
  else
    NONE

val badColonParm =
 (endsColon <$>? name <* sexp)
 errorAtEnd
 (fn s => ["there must be a space between parameter ", s,
           " and the colon that follows it"])
```

And now the formal parameters. Function `lformals` parses the formal parameters to `lambda`, and `tformals` parses the formal parameters to `type-lambda`.

S415c. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩+≡ (S413b) ◁S415b S416a▷

```
formal : (name * tyex) parser
unbracketedFormal : (name * tyex) list parser
lformals : (name * tyex) list parser
tformals : name list parser
```

```
val formal =
  bracket ("[x : ty]", badColonParm <|> pair <$> name <* kw ":" <*> ty)

val unbracketedFormal =
  (name <* kw ":" <* ty)
  errorAtEnd
  (fn x => ["the formal parameter ", x, " and its type must be ",
            "wrapped in brackets to make a list of length 1"])

val lformals = bracket ("([x : ty] ...)", unbracketedFormal <|> many formal)
val tformals = bracket ("('a ...)", many tyvar)
```

The expression parser rejects any keyword that is normally used to write types.

S416a. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩+≡ (S413b) ◁S415c S417a▷

```
fun lambda xs exp =
      nodupsty ("formal parameter", "lambda") xs >>=+ (fn xs =>
      LAMBDA (xs, exp))
fun tylambda a's exp =
      nodups ("formal type parameter", "type-lambda") a's >>=+ (fn a's =>
      TYLAMBDA (a's, exp))

fun cb key usage parser = bracketKeyword (eqx key namelike, usage, parser)

fun exp tokens = (
     VAR               <$> name
 <|> LITERAL <$> NUM   <$> int
 <|> LITERAL <$> BOOLV <$> booltok
 <|> quote *> (LITERAL <$> sexp)
 <|> quote *> badRight "quote mark ' followed by right bracket"
 <|> cb "quote"  "(quote sx)"               (       LITERAL <$> sexp)
 <|> cb "if"     "(if e1 e2 e3)"            (curry3 IFX     <$> exp  <*> exp <*> exp)
 <|> cb "while"  "(while e1 e2)"            (curry  WHILEX  <$> exp  <*> exp)
 <|> cb "set"    "(set x e)"                (curry  SET     <$> name <*> exp)
 <|> cb "begin"  ""                         (       BEGIN   <$> many exp)
 <|> cb "lambda" "(lambda (formals) body)"  (       lambda  <$> @@ lformals <*>! exp)
 <|> cb "type-lambda" "(type-lambda (tyvars) body)"
                                            (       tylambda <$> @@ tformals <*>! exp)
 <|> cb "let"    "(let (bindings) body)"    (letx   LET     <$> @@ bindings <*>! exp)
 <|> cb "letrec" "(letrec (bindings) body)" (curry  LETRECX <$> letrecbs <*> exp)
 <|> cb "let*"   "(let* (bindings) body)"   (letx   LETSTAR <$> @@ bindings <*>! exp)
 <|> cb "@"      "(@ exp types)"            (curry  TYAPPLY <$> exp <*> many1 ty)
 <|> badTyKeyword <$>! left *> @@ tyKeyword <* matchingRight
 <|> leftCurly <!> "curly brackets are not supported"
 <|> left *> right <!> "empty application"
 <|> bracket ("function application", curry APPLY <$> exp <*> many exp)
) tokens

and letx kind bs exp = letDups kind bs >>=+ (fn bs => LETX (kind, bs, exp))
and tybindings ts = bindingsOf "([x : ty] e)" formal exp ts
and letrecbs ts = distinctTBsIn (bindingsOf "([x : ty] e)" formal (asLambda "letrec" exp))
                                "letrec"
                                ts
and bindings ts = bindingsOf "(x e)" name exp ts

and badTyKeyword (loc, bad) =
      synerrorAt ("looking for expression but found `" ^ bad ^ "'") loc
```

Each `letrec` form has to have mutually distinct names. That requirement is enforced by function `distinctTBsIn`.

S416b. ⟨*parser builders for typed languages* S414c⟩+≡ (S413b S399c) ◁S414d

```
distinctTBsIn : ((name * 't) * 'e) list parser -> string ->
                ((name * 't) * 'e) list parser
```

```
fun distinctTBsIn tbindings context =
  let fun check (loc, bs) =
        nodups ("bound name", context) (loc, map (fst o fst) bs) >>=+ (fn _ => bs)
  in  check <$>! @@ tbindings
  end
```

The `def` parser handles the true-definition syntactic forms `define`, `val`, and `val-rec`.

S417a. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩+≡ (S413b) ◁S416a S417b▷

```
fun define tau f formals body =
  nodupsty ("formal parameter", "definition of function " ^ f) formals >>=+
  (fn xts =>
    DEFINE (f, tau, (xts, body)))

fun valrec (x, tau) e = VALREC (x, tau, e)

val def =
      cb "define" "(define type f (args) body)"
                            (define <$> ty <*> name <*> @@ lformals <*>! exp)
 <|> cb "val" "(val x e)" (curry VAL <$> name <*> exp)
 <|> cb "val-rec" "(val-rec [x : type] e)"
                            (valrec <$> formal <*> asLambda "val-rec" exp)
```

The `unit_test` parser handles the unit tests.

S417b. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩+≡ (S413b) ◁S417a S417c▷

```
unit_test : unit_test parser
```

```
val unit_test =
      cb "check-assert" "(check-assert e)"     (CHECK_ASSERT <$> exp)
  <|> cb "check-error"  "(check-error e)"      (CHECK_ERROR  <$> exp)
  <|> cb "check-expect" "(check-expect e1 e2)"
                                  (curry CHECK_EXPECT <$> exp <*> exp)
  <|> cb "check-type"   "(check-type e tau)"
                                  (curry CHECK_TYPE   <$> exp <*> ty)
  <|> cb "check-type-error" "(check-type-error e)"
                                  (CHECK_TYPE_ERROR <$> (def <|> EXP <$> exp))
```

And the `xdef` parser handles the extended definitions.

S417c. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩+≡ (S413b) ◁S417b S417d▷

```
xdef : xdef parser
```

```
val xdef =
     DEF <$> def
 <|> cb "use" "(use filename)" (USE <$> name)
 <|> TEST <$> unit_test
 <|> badRight "unexpected right bracket"
 <|> DEF <$> EXP <$> exp
 <?> "definition"
```

S417d. ⟨*parsers and* `xdef` *streams for Typed μScheme* S414b⟩+≡ (S413b) ◁S417c

```
val xdefstream = interactiveParsedStream (schemeToken, xdef)
```

Q.7 An infinite stream of type variables

The stream `infiniteTyvars` is used to rename type variables when checking type equality (Chapter 6). It is built from stream `naturals`, which contains the natural numbers; `naturals` is defined in chunk S230c in Appendix H.

S417e. ⟨*infinite supply of type variables* S417e⟩≡ (370a)

```
naturals       : int stream
infiniteTyvars : name stream
```

```
val infiniteTyvars =
  streamMap (fn n => "'b-" ^ intString n) naturals
```

Appendix R contents

Supporting code for nano-ML

R

R.1 Organizing code chunks into an interpreter

The overall structure of the nano-ML interpreter resembles the structure of the Typed μScheme interpreter, but instead of type checking, it uses type inference.

S419a. ⟨*ml.sml* S419a⟩≡

```
⟨exceptions used in languages with type inference S213d⟩
⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization S213a⟩

⟨Hindley-Milner types with named type constructors S420a⟩

⟨abstract syntax and values for nano-ML S419b⟩
⟨utility functions on values (μScheme, Typed μScheme, nano-ML) S422d⟩

⟨type inference for nano-ML and μML S420b⟩

⟨lexical analysis and parsing for nano-ML, providing filexdefs and stringsxdefs S432d⟩

⟨evaluation, testing, and the read-eval-print loop for nano-ML S420d⟩

⟨implementations of nano-ML primitives and definition of initialBasis S425c⟩
⟨function runStream, which evaluates input given initialBasis S240b⟩
⟨look at command-line arguments, then run S240c⟩
```

The `exp` and `value` forms are specialized to nano-ML; as usual, `xdef` and `valueString` are shared with other languages.

S419b. ⟨*abstract syntax and values for nano-ML* S419b⟩≡ (S419a)

```
⟨definitions of exp and value for nano-ML 404⟩
⟨definition of def for nano-ML 405a⟩
⟨definition of unit_test for languages with Hindley-Milner types S419c⟩
⟨definition of xdef (shared) S214b⟩
⟨definition of valueString for μScheme, Typed μScheme, and nano-ML 307b⟩
⟨definition of expString for nano-ML and μML S432a⟩
⟨definitions of defString and defName for nano-ML and μML S432c⟩
```

Unit tests resemble the unit tests for Typed μScheme, but as explained in Section 7.4.6 (page 417), the typing tests are subtly different. These unit tests are shared with other languages that use Hindley-Milner types.

S419c. ⟨*definition of* `unit_test` *for languages with Hindley-Milner types* S419c⟩≡ (S419b)

```
datatype unit_test = CHECK_EXPECT      of exp * exp
                   | CHECK_ASSERT      of exp
                   | CHECK_ERROR       of exp
                   | CHECK_TYPE        of exp * type_scheme
                   | CHECK_PTYPE       of exp * type_scheme
                   | CHECK_TYPE_ERROR  of def
```

The code chunks that make up type inference are divided into two groups. The first group contains a simpler infrastructure for Hindley-Milner types; this infrastructure works only under the assumption that each type constructor is represented by its name. This assumption holds for nano-ML, but in μML, a programmer can define two distinct type constructors that have the same name, so μML uses a less simple infrastructure that is written in a different group of code chunks.

S420a. ⟨*Hindley-Milner types with named type constructors* S420a⟩≡ (S419a)

```
⟨tycon, eqTycon, and tyconString for named type constructors 409a⟩
⟨representation of Hindley-Milner types 408⟩
⟨sets of free type variables in Hindley-Milner types 433a⟩
val funtycon = "function"
⟨creation and comparison of Hindley-Milner types with named type constructors 412b⟩
⟨definition of typeString for Hindley-Milner types S431b⟩
⟨shared utility functions on Hindley-Milner types S431d⟩
⟨specialized environments for type schemes 435a⟩
```

The second group of chunks implements type inference, and it is shared by both nano-ML and μML.

S420b. ⟨*type inference for nano-ML and μML* S420b⟩≡ (S419a)

```
typeof  : exp * type_env -> ty * con
typdef : def * type_env -> type_env * string
```

```
⟨representation of type constraints 436a⟩
⟨utility functions on type constraints S420c⟩
⟨constraint solving 437a⟩
⟨utility functions for μML S435c⟩
⟨exhaustiveness analysis for μML S435b⟩
⟨definitions of typeof and typdef for nano-ML and μML 437d⟩
```

Chapter 7 defines a handful of utility functions that operate on type-equality constraints. Two more such functions, `constraintString` and `untriviate`, are defined in this appendix.

S420c. ⟨*utility functions on type constraints* S420c⟩≡ (S420b)

```
constraintString : con -> string
untriviate       : con -> con
```

```
⟨definitions of constraintString and untriviate S431c⟩
```

On a gross level, the evaluator for nano-ML is structured in the same way as the evaluator for μScheme. Both evaluators include the usual definitions of `eval`, `evaldef`, `basis`, and `processDef`. Language-specific testing code appears in this appendix (Section R.3), and other code is shared.

S420d. ⟨*evaluation, testing, and the read-eval-print loop for nano-ML* S420d⟩≡ (S419a)

```
⟨definition of namedValueString for functional bridge languages S413a⟩
⟨definitions of eval and evaldef for nano-ML and μML S429a⟩
⟨definitions of basis and processDef for nano-ML S421a⟩
⟨shared definition of withHandlers S239a⟩
⟨shared unit-testing utilities S225a⟩
⟨definition of testIsGood for nano-ML S426a⟩
⟨shared definition of processTests S226⟩
⟨shared read-eval-print loop S237⟩
```

As in Typed Impcore and Typed μScheme, a definition is first typechecked and then evaluated. The type checker and evaluator produce strings that respectively represent type and value. When echoing, function `processDef` prints both strings. If definition `d` is not well typed, calling `typdef` raises the `TypeError` exception, and `evaldef` is never called.

S421a. ⟨*definitions of* `basis` *and* `processDef` *for nano-ML* S421a⟩≡ (S420d)

```
                            processDef : def * basis * interactivity -> basis
  type basis = type_env * value env
  fun processDef (d, (Gamma, rho), interactivity) =
    let val (Gamma, tystring)  = typdef  (d, Gamma)
        val (rho,   valstring) = evaldef (d, rho)
        val _ = if echoes interactivity then
                  println (valstring ^ " : " ^ tystring)
                else
                  ()
    in  (Gamma, rho)
    end
```

As in Typed μScheme, `processDef` preserves the phase distinction: type inference is independent of `rho` and `evaldef`.

The read-eval-print loop is almost identical to the read-eval-print loop for Typed μScheme; the only difference is that instead of a handler for `BugInTypeChecking`, it has a handler for `BugInTypeInference`.

S421b. ⟨*other handlers that catch non-fatal exceptions and pass messages to* `caught` S421b⟩≡

```
  | TypeError          msg => caught ("type error <at loc>: " ^ msg)
  | BugInTypeInference msg => caught ("bug in type inference: " ^ msg)
```

S421c. ⟨*more handlers for* `atLoc` S421c⟩≡

```
  | e as TypeError _          => raise Located (loc, e)
  | e as BugInTypeInference _ => raise Located (loc, e)
```

R.2 Primitive functions, predefined functions, and the initial basis

R.2.1 Primitives

The utility functions `unaryOp` and `binaryOp` have to be redefined yet again, because if a primitive is called with the wrong number of arguments, they must raise `BugInTypeInference`, not `BugInTypeChecking` or `RuntimeError`.

S421d. ⟨*shared utility functions for building primitives in languages with type inference* S421d⟩≡ (S4

```
                  unaryOp  : (value         -> value) -> (value list -> value)
                  binaryOp : (value * value -> value) -> (value list -> value)
  fun binaryOp f = (fn [a, b] => f (a, b)
                     | _ => raise BugInTypeInference "arity 2")
  fun unaryOp  f = (fn [a]    => f  a
                     | _ => raise BugInTypeInference "arity 1")
```

Arithmetic primitives expect and return integers.

S421e. ⟨*shared utility functions for building primitives in languages with type inference* S421d⟩+≡ (S

```
                       arithOp   : (int * int -> int) -> (value list -> value)
                       arithtype : ty
  fun arithOp f =
        binaryOp (fn (NUM n1, NUM n2) => NUM (f (n1, n2))
                   | _ => raise BugInTypeInference "arithmetic on non-numbers")
  val arithtype = funtype ([inttype, inttype], inttype)
```

Each primitive operation must be associated with a type scheme in the initial environment. It is easier, however, to associate a *type* with each primitive and to generalize the types when the initial type environment is created.

The arithmetic primitives are unsurprising:

S422a. ⟨*primitives for nano-ML and μML* **::** S422a⟩≡ (S425c) S422c ▷

```
("+", arithOp op +,   arithtype) ::
("-", arithOp op -,   arithtype) ::
("*", arithOp op *,   arithtype) ::
("/", arithOp op div, arithtype) ::
```

Nano-ML's comparison primitives take two arguments each. Some comparisons apply only to integers. The supporting functions reuse `embedBool`.

S422b. ⟨*utility functions for building nano-ML primitives* S422b⟩≡ (S425c)

```
comparison : (value * value -> bool) -> (value list -> value)
intcompare : (int   * int   -> bool) -> (value list -> value)
comptype   : ty -> ty
```

```
fun comparison f = binaryOp (embedBool o f)
fun intcompare f =
      comparison (fn (NUM n1, NUM n2) => f (n1, n2)
                   | _ => raise BugInTypeInference "comparing non-numbers")
fun comptype x = funtype ([x, x], booltype)
```

The predicates are similar to μScheme predicates. As in μScheme, values of any type can be compared for equality. Equality has type $\alpha \times \alpha \rightarrow$ `bool`, which gets generalized to type scheme $\forall\alpha \,.\, \alpha \times \alpha \rightarrow$ `bool`. As a consequence; nano-ML's type system permits code to compare functions for equality. (If the code is evaluated, it causes a checked run-time error.) In Standard ML, by contrast, the type system prevents code from comparing values of function types.

S422c. ⟨*primitives for nano-ML and μML* **::** S422a⟩+≡ (S425c) ◁ S422a S423a ▷

```
("<", intcompare op <,             comptype inttype) ::
(">", intcompare op >,             comptype inttype) ::
("=", comparison primitiveEquality, comptype alpha) ::
```

S422d. ⟨*utility functions on values (μScheme, Typed μScheme, nano-ML)* S422d⟩≡ (S419a S379 S405a)

```
fun primitiveEquality (v, v') =
 let fun noFun () = raise RuntimeError "compared functions for equality"
 in  case (v, v')
       of (NIL,      NIL    )  => true
        | (NUM  n1,  NUM  n2)  => (n1 = n2)
        | (SYM  v1,  SYM  v2)  => (v1 = v2)
        | (BOOLV b1, BOOLV b2) => (b1 = b2)
        | (PAIR (v, vs), PAIR (v', vs')) =>
            primitiveEquality (v, v') andalso primitiveEquality (vs, vs')
        | (PAIR _,   NIL)    => false
        | (NIL,      PAIR _) => false
        | (CLOSURE   _, _) => noFun ()
        | (PRIMITIVE _, _) => noFun ()
        | (_, CLOSURE   _) => noFun ()
        | (_, PRIMITIVE _) => noFun ()
        | _ => raise BugInTypeInference
                     ("compared incompatible values " ^ valueString v ^
                      " and " ^ valueString v' ^ " for equality")
 end
```

Primitives `print` and `println` are polymorphic.

S423a. ⟨*primitives for nano-ML and μML* `::` S422a⟩+≡ (S425c) ◁S422c

```
("println", unaryOp (fn v => (print (valueString v ^ "\n"); v)),
               funtype ([alpha], unittype)) ::
("print",   unaryOp (fn v => (print (valueString v);        v)),
               funtype ([alpha], unittype)) ::
("printu",  unaryOp (fn NUM n => (printUTF8 n; NUM n)
                       | _ => raise BugInTypeInference "printu of non-number"),
               funtype ([inttype], unittype)) ::
```

R.2.2 Predefined functions

Binding a pair into an association list works just as in untyped μScheme.

S423b. ⟨*predefined nano-ML functions* S423b⟩≡ S423c ▷

```
(define bind (x y alist)
  (if (null? alist)
    (cons (pair x y) '())
    (if (= x (fst (car alist)))
      (cons (pair x y) (cdr alist))
      (cons (car alist) (bind x y (cdr alist))))))
```

But search is different. When `find` is asked about an unbound variable, it can't return the empty list, because the empty list is not always of the right type. Looking up an unbound variable must therefore be a checked run-time error. To see if a variable is unbound, To avoid such errors, nano-ML code must use the predicate `bound?`. (A good alternative would be to use continuation-passing style, as in Chapter 2.)

S423c. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁S423b S423d ▷

```
(define find (x alist)
  (if (null? alist)
    (error 'not-found)
    (if (= x (fst (car alist)))
      (snd (car alist))
      (find x (cdr alist)))))
(define bound? (x alist)
  (if (null? alist)
    #f
    (if (= x (fst (car alist)))
      #t
      (bound? x (cdr alist)))))
```

The remaining predefined functions are identical to their counterparts in μScheme.

S423d. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁S423c S423e ▷

```
(define caar (xs) (car (car xs)))
(define cadr (xs) (car (cdr xs)))
(define cdar (xs) (cdr (car xs)))
(define and (b c) (if b  c  b))
(define or  (b c) (if b  b  c))
(define not (b)   (if b #f #t))
```

S423e. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁S423d S424a ▷

```
(define append (xs ys)
  (if (null? xs)
     ys
     (cons (car xs) (append (cdr xs) ys))))
```

S424a. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁ S423e S424b ▷

```
(define revapp (xs ys)
  (if (null? xs)
     ys
     (revapp (cdr xs) (cons (car xs) ys))))
(define reverse (xs) (revapp xs '()))
```

S424b. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁ S424a S424c ▷

```
(define o (f g) (lambda (x) (f (g x))))
(define curry   (f) (lambda (x) (lambda (y) (f x y))))
(define uncurry (f) (lambda (x y) ((f x) y)))
```

S424c. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁ S424b S424d ▷

```
(define filter (p? xs)
  (if (null? xs)
    '()
    (if (p? (car xs))
      (cons (car xs) (filter p? (cdr xs)))
      (filter p? (cdr xs)))))
```

S424d. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁ S424c S424e ▷

```
(define map (f xs)
  (if (null? xs)
    '()
    (cons (f (car xs)) (map f (cdr xs)))))
```

S424e. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁ S424d S424f ▷

```
(define exists? (p? xs)
  (if (null? xs)
    #f
    (if (p? (car xs))
      #t
      (exists? p? (cdr xs)))))
(define all? (p? xs)
  (if (null? xs)
    #t
    (if (p? (car xs))
      (all? p? (cdr xs))
      #f)))
```

S424f. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁ S424e S424g ▷

```
(define foldr (op zero xs)
  (if (null? xs)
    zero
    (op (car xs) (foldr op zero (cdr xs)))))
(define foldl (op zero xs)
  (if (null? xs)
    zero
    (foldl op (op (car xs) zero) (cdr xs))))
```

S424g. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁ S424f S424h ▷

```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

S424h. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁ S424g S425a ▷

```
(define max (x y) (if (> x y) x y))
(define min (x y) (if (< x y) x y))
(define negated (n) (- 0 n))
(define mod (m n) (- m (* n (/ m n))))
(define gcd (m n) (if (= n 0) m (gcd n (mod m n))))
(define lcm (m n) (* m (/ n (gcd m n))))
```

S425a. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁S424h S425b▷

```
(define min* (xs) (foldr min (car xs) (cdr xs)))
(define max* (xs) (foldr max (car xs) (cdr xs)))
(define gcd* (xs) (foldr gcd (car xs) (cdr xs)))
(define lcm* (xs) (foldr lcm (car xs) (cdr xs)))
```

S425b. ⟨*predefined nano-ML functions* S423b⟩+≡ ◁S425a

```
(define list1 (x)               (cons x '()))
(define list2 (x y)             (cons x (list1 y)))
(define list3 (x y z)           (cons x (list2 y z)))
(define list4 (x y z a)         (cons x (list3 y z a)))
(define list5 (x y z a b)       (cons x (list4 y z a b)))
(define list6 (x y z a b c)     (cons x (list5 y z a b c)))
(define list7 (x y z a b c d)   (cons x (list6 y z a b c d)))
(define list8 (x y z a b c d e) (cons x (list7 y z a b c d e)))
```

R.2.3 Building the initial basis

Given primitives and user code, the interpreter computes the type environment and the value environment simultaneously.

S425c. ⟨*implementations of nano-ML primitives and definition of* `initialBasis` S425c⟩≡ (S419a)

```
initialBasis : type_env * value env
```

```
⟨shared utility functions for building primitives in languages with type inference S421d⟩
⟨utility functions for building nano-ML primitives S422b⟩
val primitiveBasis =
  let fun addPrim ((name, prim, tau), (Gamma, rho)) =
        ( bindtyscheme (name, generalize (tau, freetyvarsGamma Gamma), Gamma)
        , bind (name, PRIMITIVE prim, rho)
        )
  in  foldl addPrim (emptyTypeEnv, emptyEnv)
                    (⟨primitives for nano-ML and μML :: S422a⟩
                     ⟨primitives for nano-ML :: 440a⟩
                     [])
  end
val predefs = ⟨predefined nano-ML functions, as strings (from ⟨predefined nano-ML functions S423b⟩)⟩
val initialBasis =
  let val xdefs = stringsxdefs ("predefined functions", predefs)
  in  readEvalPrintWith predefinedFunctionError
        (xdefs, primitiveBasis, noninteractive)
  end
```

bind	305d
bindtyscheme	435c
emptyEnv	305a
emptyTypeEnv	435b
type env	304
freetyvarsGamma	435d
generalize	434b
noninteractive	S236c
predefined-Function-Error	S215c
PRIMITIVE	405b
readEvalPrintWith	S237
stringsxdefs	S233a
type type_env	435a
type value	405b

R.3 Unit testing

Overall, function `testIsGood` is structured in the same way as in the interpreters for Typed Impcore and Typed μScheme, but the details of the testing are quite different: nano-ML offers not only `check-type` but also `check-principal-type`.

S426a. ⟨*definition of* `testIsGood` *for nano-ML* S426a⟩≡ (S420d)

```
⟨definition of skolemTypes for languages with named type constructors S427b⟩
⟨shared definitions of typeSchemeIsAscribable and typeSchemeIsEquivalent S427c⟩
fun testIsGood (test, (Gamma, rho)) =
  let fun ty e = typeof (e, Gamma)
                 handle NotFound x =>
                   raise TypeError ("name " ^ x ^ " is not defined")
      fun deftystring d =
        snd (typdef (d, Gamma))
        handle NotFound x => raise TypeError ("name " ^ x ^ " is not defined")
      ⟨definitions of check{Expect,Assert,Error{Checks that use type inference S426b⟩
      ⟨definitions of check{Expect,Assert,Error{Checks that use type inference S426b⟩
      ⟨definition of checkTypeChecks using type inference S427a⟩
      fun checks (CHECK_EXPECT (e1, e2)) = checkExpectChecks (e1, e2)
        | checks (CHECK_ASSERT e)        = checkAssertChecks e
        | checks (CHECK_ERROR e)         = checkErrorChecks e
        | checks (CHECK_TYPE  (e, tau))  = checkTypeChecks "check-type" (e, tau)
        | checks (CHECK_PTYPE (e, tau))  = checkTypeChecks "check-principal-type"
                                                                       (e, tau)
        | checks (CHECK_TYPE_ERROR e)    = true

      fun outcome e = withHandlers (fn () => OK (eval (e, rho))) () (ERROR o stripAtLoc)
      ⟨asSyntacticValue for μScheme, Typed Impcore, Typed μScheme, and nano-ML S383a⟩
      ⟨shared check{Expect,Assert,Error{Passes, which call outcome S224d⟩
      ⟨definitions of check*Type*Passes using type inference S428b⟩
      fun passes (CHECK_EXPECT (c, e))  = checkExpectPasses (c, e)
        | passes (CHECK_ASSERT c)       = checkAssertPasses c
        | passes (CHECK_ERROR c)        = checkErrorPasses  c
        | passes (CHECK_TYPE  (c, tau)) = checkTypePasses          (c, tau)
        | passes (CHECK_PTYPE (c, tau)) = checkPrincipalTypePasses (c, tau)
        | passes (CHECK_TYPE_ERROR c)   = checkTypeErrorPasses c

  in  checks test andalso passes test
  end
```

S426b. ⟨*definitions of* `check{Expect,Assert,Error{Checks` *that use type inference* S426b⟩≡ (S426a) S426c ▷

```
fun checkExpectChecks (e1, e2) =
  let val (tau1, c1) = ty e1
      val (tau2, c2) = ty e2
      val c = tau1 ~ tau2
      val theta = solve (c1 /\ c2 /\ c)
  in  true
  end handle TypeError msg =>
      failtest ["In (check-expect ", expString e1, " ", expString e2, "), ", msg]
```

S426c. ⟨*definitions of* `check{Expect,Assert,Error{Checks` *that use type inference* S426b⟩+≡ (S426a) ◁ S426b

```
fun checkExpChecksIn what e =
  let val (tau, c) = ty e
      val theta = solve c
  in  true
  end handle TypeError msg =>
      failtest ["In (", what, " ", expString e, "), ", msg]
val checkAssertChecks = checkExpChecksIn "check-assert"
val checkErrorChecks  = checkExpChecksIn "check-error"
```

S427a. ⟨*definition of* checkTypeChecks *using type inference* S427a⟩≡ (S426a)

```
fun checkTypeChecks form (e, sigma) =
  let fun fail msg =
        failtest ["In (", form, " ", expString e, " " ^ typeSchemeString sigma, "), ", msg]
      fun freevars (FORALL (alphas, tau)) = diff (freetyvars tau, alphas)
      fun unused   (FORALL (alphas, tau)) = diff (alphas, freetyvars tau)
  in  let val (tau, c) = ty e
          val theta  = solve c
      in  case (freevars sigma, unused sigma)
            of ([], []) => true
             | (alpha :: _, _) => fail ("type variable " ^ alpha ^ " must be qua
             | (_, alpha :: _) => fail ("quantified type variable " ^ alpha ^ "
      end handle TypeError msg => fail msg
  end
```

The logic needed to implement check-type is not so simple. A check-type needs to pass if the type given in the test is an *instance* of the principal type of the thing tested—that is, the type of the thing tested can be more polymorphic than what the test is asking for. The instance property is not so easy to check directly—searching for permutations is tedious—but the idea is simple: no matter what types are used to instantiate σ_i, σ_g can be instantiated to the same type. To help myself implement this idea, I define a supply of *skolem types* that cannot possibly be part of any type in any nano-ML program.

S427b. ⟨*definition of* skolemTypes *for languages with named type constructors* S427b⟩≡ (S426a)

```
                                                  skolemTypes : ty stream
val skolemTypes =
  streamMap (fn n => TYCON ("skolem type " ^ intString n)) naturals
```

Skolem types are used to create an "arbitrary" instance of type scheme σ_i. If the constraint solver can make that instance equal to a fresh instance of σ_g, then σ_g is as general as σ_i.

S427c. ⟨*shared definitions of* typeSchemeIsAscribable *and* typeSchemeIsEquivalent S427c⟩≡ (S

```
                               asGeneralAs : type_scheme * type_scheme -> bool
fun asGeneralAs (sigma_g, sigma_i as FORALL (a's, tau)) =
  let val theta = mkEnv (a's, streamTake (length a's, skolemTypes))
      val skolemized = tysubst theta tau
      val tau_g = freshInstance sigma_g
  in  (solve (tau_g ~ skolemized); true) handle _ => false
  end
```

Function asGeneralAs suffices to implement the check-type test. The test passes if the type of e is as general as the type being claimed for e.

S427d. ⟨*shared definitions of* typeSchemeIsAscribable *and* typeSchemeIsEquivalent S427c⟩+≡

```
fun typeSchemeIsAscribable (e, sigma_e, sigma) =
  if asGeneralAs (sigma_e, sigma) then
    true
  else
    failtest ["check-type failed: expected ", expString e,
              " to have type ", typeSchemeString sigma,
              ", but it has type ", typeSchemeString sigma_e]
```

And asGeneralAs is also sufficient to implement check-principal-type, which checks for equivalence. Two type schemes are equivalent if each is as general as the other. To avoid having to write error messages twice, I implement one of the generality checks using typeSchemeIsAscribable.

S428a. ⟨*shared definitions of* typeSchemeIsAscribable *and* typeSchemeIsEquivalent S427c⟩+≡ (S426a) ◁ S42

```
fun typeSchemeIsEquivalent (e, sigma_e, sigma) =
  if typeSchemeIsAscribable (e, sigma_e, sigma) then
    if asGeneralAs (sigma, sigma_e) then
      true
    else
      failtest ["check-principal-type failed: expected ", expString e,
                " to have principal type ", typeSchemeString sigma,
                ", but it has the more general type ", typeSchemeString sigma_e]
  else
    false  (* error message already issued *)
```

Each unit test first computes sigma_e, then calls the appropriate function.

S428b. ⟨*definitions of* check*Type*Passes *using type inference* S428b⟩≡ (S426a) S428c ▷

```
fun checkTypePasses (e, sigma) =
  let val (tau, c) = ty e
      val theta    = solve c
      val sigma_e  = generalize (tysubst theta tau, freetyvarsGamma Gamma)
  in  typeSchemeIsAscribable (e, sigma_e, sigma)
  end handle TypeError msg =>
      failtest ["In (check-type ", expString e,
                " ", typeSchemeString sigma, "), ", msg]
```

S428c. ⟨*definitions of* check*Type*Passes *using type inference* S428b⟩+≡ (S426a) ◁ S428b S428d ▷

```
fun checkPrincipalTypePasses (e, sigma) =
  let val (tau, c) = ty e
      val theta    = solve c
      val sigma_e  = generalize (tysubst theta tau, freetyvarsGamma Gamma)
  in  typeSchemeIsEquivalent (e, sigma_e, sigma)
  end handle TypeError msg =>
      failtest ["In (check-principal-type ", expString e, " ",
                typeSchemeString sigma, "), ", msg]
```

The check-type-error tests expects a type error while computing sigma_e.

S428d. ⟨*definitions of* check*Type*Passes *using type inference* S428b⟩+≡ (S426a) ◁ S428c

```
fun checkTypeErrorPasses (EXP e) =
      (let val (tau, c) = ty e
           val theta  = solve c
           val sigma' = generalize (tysubst theta tau, freetyvarsGamma Gamma)
       in  failtest ["check-type-error failed: expected ", expString e,
                     " not to have a type, but it has type ",
                     typeSchemeString sigma']
       end handle TypeError msg => true
                | Located (_, TypeError _) => true)
  | checkTypeErrorPasses d =
      (let val t = deftystring d
       in  failtest ["check-type-error failed: expected ", defString d,
                     " to cause a type error, but it successfully defined ",
                     defName d, " : ", t
                    ]
       end handle TypeError msg => true
                | Located (_, TypeError _) => true)
```

R.4 Evaluation

R.4.1 *Evaluation of expressions*

Syntactically, the nano-ML expressions are a subset of the μScheme expressions. Therefore, the nano-ML evaluator is almost a subset of the μScheme evaluator. But because nano-ML doesn't have mutation, environments map names to values, instead of mapping them to mutable cells. And fewer errors should be possible at evaluation time, because type inference should rule them out. If one of those errors occurs anyway, the evaluator raises the exception `BugInTypeInference`.

S429a. ⟨*definitions of* `eval` *and* `evaldef` *for nano-ML and μML* S429a⟩≡ (S420d) S430b ▷

```
eval : exp * value env -> value
```

```
fun eval (e, rho) =
  let fun ev (LITERAL v)        = v
        | ev (VAR x)            = find (x, rho)
        | ev (IFX (e1, e2, e3)) = ev (if projectBool (ev e1) then e2 else e3)
        | ev (LAMBDA l)         = CLOSURE (l, ref rho)
        | ev (BEGIN es) =
            let fun b (e::es, lastval) = b (es, ev e)
                  | b (   [], lastval) = lastval
            in  b (es, embedBool false)
            end
        | ev (APPLY (f, args)) =
           (case ev f
              of PRIMITIVE prim => prim (map ev args)
               | CLOSURE clo => ⟨apply closure clo to args S429b⟩
               | _ => raise BugInTypeInference "Applied non-function"
               )
        ⟨more alternatives for ev for nano-ML and μML S429c⟩
  in  ev e
  end
```

To apply a closure, the evaluator binds formal parameters directly to the values of actual parameters, not to mutable cells. Environment ρ_c is extended with the formal-parameter bindings using the <+> function.

S429b. ⟨*apply closure* `clo` *to* `args` S429b⟩≡ (S429a)

```
let val ((formals, body), ref rho_c) = clo
    val actuals = map ev args
in  eval (body, rho_c <+> mkEnv (formals, actuals))
    handle BindListLength =>
        raise BugInTypeInference "Wrong number of arguments to closure"
end
```

LET evaluates all right-hand sides in ρ, then extends ρ to evaluate the body.

S429c. ⟨*more alternatives for* `ev` *for nano-ML and μML* S429c⟩≡ (S429a) S429d ▷

```
| ev (LETX (LET, bs, body)) =
    let val (names, values) = ListPair.unzip bs
    in  eval (body, rho <+> mkEnv (names, map ev values))
    end
```

LETSTAR evaluates pairs in sequence, adding a binding to ρ after each evaluation.

S429d. ⟨*more alternatives for* `ev` *for nano-ML and μML* S429c⟩+≡ (S429a) ◁ S429c S430a ▷

```
| ev (LETX (LETSTAR, bs, body)) =
    let fun step ((x, e), rho) = bind (x, eval (e, rho), rho)
    in  eval (body, foldl step rho bs)
    end
```

LETREC is the most interesting case. Function `makeRho'` builds an environment in which each right-hand side stands for a closure. Each closure's captured environment is the one built by `makeRho'`. The recursion is OK because the environment is built lazily, so `makeRho'` always terminates. The parser guarantees that the right-hand sides are lambda abstractions.

S430a. ⟨*more alternatives for* `ev` *for nano-ML and* μ*ML* S429c⟩+≡ (S429a) ◁S429d

```
  | ev (LETX (LETREC, bs, body)) =
      let fun asLambda (LAMBDA l) = l
            | asLambda _ = raise InternalError "parser guaranteed lambda"
          val newref = ref emptyEnv
          val rho' = foldl (fn ((x, e), rho) =>
                              bind (x, CLOSURE (asLambda e, newref), rho))
                          rho
                          bs
          val () = newref := rho'
      in  eval (body, rho')
      end
```

R.4.2 Evaluation of definitions

Evaluating a definition can produce a new environment. Function `evaldef` also returns a string that identifies the name or value being defined.

S430b. ⟨*definitions of* `eval` *and* `evaldef` *for nano-ML and* μ*ML* S429a⟩+≡ (S420d) ◁S429a S430c▷

```
evaldef : def * value env -> value env * string
```

```
  fun evaldef (VAL (x, e), rho) =
        let val v   = eval (e, rho)
            val rho = bind (x, v, rho)
        in  (rho, namedValueString x v)
        end
    | evaldef (VALREC (f, LAMBDA lambda), rho) =
        let val newref = ref emptyEnv
            val rho = bind (f, CLOSURE (lambda, newref), rho)
            val () = newref := rho
        in  (rho, f)
        end
    | evaldef (VALREC _, rho) =
        raise InternalError "expression in val-rec is not lambda"
    | evaldef (EXP e, rho) =
        let val v   = eval (e, rho)
            val rho = bind ("it", v, rho)
        in  (rho, valueString v)
        end
```

The implementation of `VALREC` works only for `LAMBDA` expressions because these are the only expressions whose value can be computed without having the environment.

As in the type system, `DEFINE` is syntactic sugar for a combination of `VALREC` and `LAMBDA`.

S430c. ⟨*definitions of* `eval` *and* `evaldef` *for nano-ML and* μ*ML* S429a⟩+≡ (S420d) ◁S430b

```
    | evaldef (DEFINE (f, lambda), rho) =
        evaldef (VALREC (f, LAMBDA lambda), rho)
    ⟨clause for evaldef for datatype definition (μML only) S431a⟩
```

μML, which is the subject of Chapter 8, is like nano-ML but with one additional definition form, for defining an algebraic data type. Nano-ML lacks that form, so the corresponding clause in `evaldef` is empty.

S431a. ⟨*clause for* `evaldef` *for datatype definition (μML only)* S431a⟩≡ (S430c)

```
(* code goes here in Appendix S *)
```

R.5 String conversion

Function types are printed infix, and other constructor applications are printed prefix.

S431b. ⟨*definition of* `typeString` *for Hindley-Milner types* S431b⟩≡ (S420a)

```
fun typeString tau =
  case asFuntype tau
    of SOME (args, result) =>
         "(" ^ spaceSep (map typeString args) ^ " -> " ^
               typeString result ^ ")"
     | NONE =>
         case tau
           of TYCON c => tyconString c
            | TYVAR a => a
            | CONAPP (tau, []) => "(" ^ typeString tau ^ ")"
            | CONAPP (tau, taus) =>
                "(" ^ typeString tau ^ " " ^
                      spaceSep (map typeString taus) ^ ")"
```

A constraint can be printed in full, but it's easier to read if its first passed to `untriviate`, which removes as many `TRIVIAL` sub-constraints as possible.

S431c. ⟨*definitions of* `constraintString` *and* `untriviate` S431c⟩≡ (S420c)

```
fun constraintString (c /\ c') =
      constraintString c ^ " /\\ " ^ constraintString c'
  | constraintString (t ~  t') = typeString t ^ " ~ " ^ typeString t'
  | constraintString TRIVIAL = "TRIVIAL"

fun untriviate (c /\ c') = (case (untriviate c, untriviate c')
                              of (TRIVIAL, c) => c
                               | (c, TRIVIAL) => c
                               | (c, c') => c /\ c')
  | untriviate atomic = atomic
```

A degenerate type scheme is printed as if it were a type. But when a nondegenerate polytype is printed, the `forall` is explicit, and all the quantified variables are shown.[1]

S431d. ⟨*shared utility functions on Hindley-Milner types* S431d⟩≡ (S420a)

```
typeString       : ty          -> string
typeSchemeString : type_scheme -> string
```

```
fun typeSchemeString (FORALL ([], tau)) =
      typeString tau
  | typeSchemeString (FORALL (a's, tau)) =
      "(forall [" ^ spaceSep a's ^ "] " ^ typeString tau ^ ")"
```

[1] It is not strictly necessary to show the quantified variables, because in any top-level type, *all* type variables are quantified by the ∀. For this reason, Standard ML leaves out quantifiers and type variables. But when you're learning about parametric polymorphism, explicit `forall`s are better. In my experience, Standard ML's implicit `forall`s make it hard to new learners to understand what's going on.

S432a. ⟨*definition of* `expString` *for nano-ML and μML* S432a⟩≡ (S419b)

```
fun expString e =
  let fun bracket s = "(" ^ s ^ ")"
      fun sqbracket s = "[" ^ s ^ "]"
      val bracketSpace = bracket o spaceSep
      fun exps es = map expString es
      fun withBindings (keyword, bs, e) =
        bracket (spaceSep [keyword, bindings bs, expString e])
      and bindings bs = bracket (spaceSep (map binding bs))
      and binding (x, e) = sqbracket (x ^ " " ^ expString e)
      val letkind = fn LET => "let" | LETSTAR => "let*" | LETREC => "letrec"
  in  case e
        of LITERAL v => valueString v
         | VAR name => name
         | IFX (e1, e2, e3) => bracketSpace ("if" :: exps [e1, e2, e3])
         | BEGIN es => bracketSpace ("begin" :: exps es)
         | APPLY (e, es) => bracketSpace (exps (e::es))
         | LETX (lk, bs, e) => bracketSpace [letkind lk, bindings bs, expString e]
         | LAMBDA (xs, body) => bracketSpace ["lambda",
                                              bracketSpace xs, expString body]
         ⟨extra cases of expString for μML S432b⟩
  end
```

S432b. ⟨*extra cases of* `expString` *for μML* S432b⟩≡ (S432a)

```
(* this space is filled in by the uML appendix *)
```

S432c. ⟨*definitions of* `defString` *and* `defName` *for nano-ML and μML* S432c⟩≡ (S419b)

```
fun defString d =
  let fun bracket s = "(" ^ s ^ ")"
      val bracketSpace = bracket o spaceSep
      fun formal (x, t) = "[" ^ x ^ " : " ^ typeString t ^ "]"
  in  case d
        of EXP e         => expString e
         | VAL    (x, e) => bracketSpace ["val",     x, expString e]
         | VALREC (x, e) => bracketSpace ["val-rec", x, expString e]
         | DEFINE (f, (formals, body)) =>
             bracketSpace ["define", f, bracketSpace formals, expString body]
         ⟨cases for defString for forms found only in μML (from chunk 697b)⟩
  end
fun defName (VAL (x, _))       = x
  | defName (VALREC (x, _)) = x
  | defName (DEFINE (x, _)) = x
  | defName (EXP _) = raise InternalError "asked for name defined by expression"
  ⟨clauses for defName for forms found only in μML (from chunk 697b)⟩
```

R.6 Parsing

Nano-ML uses the same lexical analysis as μScheme, and many of the same parsers.

S432d. ⟨*lexical analysis and parsing for nano-ML, providing* `filexdefs` *and* `stringsxdefs` S432d⟩≡ (S419a)

```
⟨lexical analysis for μScheme and related languages S383d⟩
⟨parsers for single tokens for μScheme-like languages S385a⟩
⟨parsers for nano-ML tokens S433c⟩
⟨parsers and parser builders for formal parameters and bindings S385c⟩
⟨parsers and parser builders for Scheme-like syntax S386d⟩
⟨parser builders for typed languages S414c⟩
⟨parsers for Hindley-Milner types with named type constructors S433d⟩
⟨parsers and xdef streams for nano-ML S433a⟩
⟨shared definitions of filexdefs and stringsxdefs S233a⟩
```

Nano-ML lacks `set` and `while`.

S433a. ⟨*parsers and* `xdef` *streams for nano-ML* S433a⟩≡ (S432d) S434b ▷

```
exp      : exp parser
exptable : exp parser -> exp parser
```

```
fun exptable exp =
  let val bindings = bindingsOf "(x e)" name exp
      val dbs      = distinctBsIn bindings
      val formals  = formalsOf "(x1 x2 ...)" name "lambda"
      val letrecbs = distinctBsIn (bindingsOf "[f (lambda (...) ...)]"
                                              name
                                              (asLambda "letrec" exp))
                                  "letrec"
  in usageParsers
     [ ("(if e1 e2 e3)",           curry3 IFX          <$> exp <*> exp <*> exp)
     , ("(begin e1 ...)",                 BEGIN        <$> many exp)
     , ("(lambda (names) body)",    curry  LAMBDA       <$> formals      <*> exp)
     , ("(let (bindings) body)",    curry3 LETX LET     <$> dbs "let"    <*> exp)
     , ("(letrec (bindings) body)", curry3 LETX LETREC  <$> letrecbs     <*> exp)
     , ("(let* (bindings) body)",   curry3 LETX LETSTAR <$> bindings     <*> exp)
     ⟨rows added to nano-ML's exptable in exercises S433b⟩
     ]
  end

val exp = fullSchemeExpOf (atomicSchemeExpOf name) exptable
```

S433b. ⟨*rows added to nano-ML's* `exptable` *in exercises* S433b⟩≡ (S433a)

```
(* add syntactic extensions here, each preceded by a comma *)
```

Nano-ML shares Typed μScheme's requirements for names. It even uses the same reserved words. (Although they aren't valid keywords in nano-ML, the parser reserves the words `set` and `while`. That reservation ensures that every grammatical nano-ML program is also a grammatical μScheme program.)

S433c. ⟨*parsers for nano-ML tokens* S433c⟩≡ (S432d)

```
val reserved = [ "if", "while", "set", "begin", "lambda", "let"
               , "letrec", "let*", "quote", "val", "define", "use"
               , "check-expect", "check-assert", "check-error"
               , "check-type", "check-principal-type", "check-type-error"
               ]

val arrow = eqx "->" namelike
val name  = rejectReserved reserved <$>! sat (curry op <> "->") namelike
val tyvar =
  quote *> (curry op ^ "'" <$> name <?> "type variable (got quote mark)")
```

Types are parsed much as in Appendix Q, from which the `arrowsOf` function is reused. But a `forall` type is not accepted as a type; it's a type scheme.

S433d. ⟨*parsers for Hindley-Milner types with named type constructors* S433d⟩≡ (S432d) S434a ▷

```
tyvar    : string      parser
ty       : ty          parser
tyscheme : type_scheme parser
```

```
val arrows = arrowsOf CONAPP funtype

fun ty tokens = (
     TYCON <$> name
 <|> TYVAR <$> tyvar
 <|> usageParsers
     [("(forall (tyvars) type)", bracket ("('a ...)", many tyvar) *> ty)]
     <!> "nested 'forall' type is not a Hindley-Milner type"
 <|> bracket ("constructor application",
              arrows <$> many ty <*>! many (arrow *> many ty))
) tokens
```

Only parser `tyscheme` accepts a `forall` type.

S434a. ⟨*parsers for Hindley-Milner types with named type constructors* S433d⟩+≡ (S432d) ◁ S433d

```
val tyscheme =
      usageParsers [("(forall (tyvars) type)",
                     curry FORALL <$> bracket ("['a ...]", distinctTyvars) <*> ty)]
  <|> curry FORALL [] <$> ty
  <?> "type"
```

True definitions and extended definitions are as expected.

S434b. ⟨*parsers and* `xdef` *streams for nano-ML* S433a⟩+≡ (S432d) ◁ S433a S434d ▷

```
deftable  : def       parser
testtable : unit_test parser
xdef      : xdef      parser
```

```
val deftable = usageParsers
  [ ("(define f (args) body)",
                  let val formals = formalsOf "(x1 x2 ...)" name "define"
                  in  curry DEFINE <$> name <*> (pair <$> formals <*> exp)
                  end)
  , ("(val x e)",     curry VAL    <$> name <*> exp)
  , ("(val-rec x e)", curry VALREC <$> name <*> asLambda "val-rec" exp)
  ]

val testtable = usageParsers
  [ ("(check-expect e1 e2)",         curry CHECK_EXPECT <$> exp <*> exp)
  , ("(check-assert e)",                   CHECK_ASSERT <$> exp)
  , ("(check-error e)",                    CHECK_ERROR  <$> exp)
  , ("(check-type e tau)",           curry CHECK_TYPE   <$> exp <*> tyscheme)
  , ("(check-principal-type e tau)", curry CHECK_PTYPE  <$> exp <*> tyscheme)
  , ("(check-type-error e)",               CHECK_TYPE_ERROR <$> (deftable <|>
                                                                 EXP <$> exp))

  ]

val xdeftable = usageParsers
  [ ("(use filename)", USE <$> name)
  ⟨rows added to nano-ML's xdeftable in exercises S434c⟩
  ]

val xdef  =  TEST <$> testtable
         <|> DEF  <$> deftable
         <|>          xdeftable
         <|> badRight "unexpected right bracket"
         <|> DEF <$> EXP <$> exp
         <?> "definition"
```

S434c. ⟨*rows added to nano-ML's* `xdeftable` *in exercises* S434c⟩≡ (S434b)

```
(* add syntactic extensions here, each preceded by a comma *)
```

S434d. ⟨*parsers and* `xdef` *streams for nano-ML* S433a⟩+≡ (S432d) ◁ S434b

```
val xdefstream = interactiveParsedStream (schemeToken, xdef)
```

R.7 Cases and code for Chapter 8

Nano-ML is the foundation for μML (Chapter 8), which adds cases for pattern matching and algebraic data types. The following code chunks are placeholders for code that is added in Chapter 8.

S435a. ⟨*extra case for* `typdef` *used only in* μ*ML* S435a⟩≡ (439a)

```
  (* filled in when implementing uML *)
```

S435b. ⟨*exhaustiveness analysis for* μ*ML* S435b⟩≡ (S420b)

```
  (* filled in when implementing uML *)
```

S435c. ⟨*utility functions for* μ*ML* S435c⟩≡ (S420b)

```
  (* filled in when implementing uML *)
```

Appendix S contents

Supporting code for μML S

S.1 Organizing code chunks into an interpreter

μML is very similar to nano-ML, and so is the organization of the interpreter.

S437a. ⟨*uml.sml* S437a⟩≡
⟨*exceptions used in languages with type inference* S213d⟩
⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S213a⟩

⟨*Hindley-Milner types with generated type constructors* S438d⟩

⟨*abstract syntax and values for μML* S437b⟩
⟨*utility functions on μML syntax* S452b⟩
⟨*utility functions on μML values* (from chunk 697b)⟩

⟨*lexical analysis and parsing for μML, providing* `filexdefs` *and* `stringsxdefs` S462e⟩

⟨*definition of* `basis` *for μML* S439a⟩
⟨*translation of μML type syntax into types* S455a⟩
⟨*typing and evaluation of* `data` *definitions* S439c⟩
⟨*definitions of* `emptyBasis` *and* `predefinedTypeBasis` S441b⟩
⟨*definitions of* `booltype`, `listtype`, *and* `unittype` S442b⟩
⟨*type inference for nano-ML and μML* S420b⟩

⟨*evaluation, testing, and the read-eval-print loop for μML* S438e⟩

⟨*implementations of μML primitives and definition of* `initialBasis` S442c⟩
⟨*function* `runStream`, *which evaluates input given* `initialBasis` S240b⟩
⟨*look at command-line arguments, then run* S240c⟩

Abstract syntax includes not only expressions and definitions but also types, patterns, and unit tests.

S437b. ⟨*abstract syntax and values for μML* S437b⟩≡ (S437a)
⟨*kinds for typed languages* S454a⟩
⟨*definition of* `tyex` *for μML* S454c⟩
⟨*definition of* `pat`, *for patterns* 486c⟩
⟨*definitions of* `exp` *and* `value` *for μML* 486b⟩
⟨*definition of* `def` *for μML* 486a⟩
⟨*definition of* `implicit_data_def` *for μML* S469c⟩
⟨*definition of* `unit_test` *for languages with Hindley-Milner types and generated type constructors* S438c⟩
⟨*definition of* `xdef` *(shared)* S214b⟩
⟨*definition of* `valueString` *for μML* S461b⟩
⟨*definition of* `patString` *for μML and μHaskell* (from chunk 697b)⟩
⟨*definition of* `expString` *for nano-ML and μML* S432a⟩
⟨*definitions of* `defString` *and* `defName` *for nano-ML and μML* S432c⟩
⟨*definition of* `tyexString` *for μML* S462c⟩

The syntactic forms that are unique to μML are shown in Chapter 8. The remaining forms, which are carried over form nano-ML, are defined here.

S438a. ⟨*forms of* `exp` *carried over from nano-ML* S438a⟩≡ (486b)

```
      LITERAL    of value
    | VAR        of name
    | IFX        of exp * exp * exp (* could be syntactic sugar for CASE *)
    | BEGIN      of exp list
    | APPLY      of exp * exp list
    | LETX       of let_flavor * (name * exp) list * exp
    | LAMBDA     of name list * exp
  and let_flavor = LET | LETREC | LETSTAR
```

S438b. ⟨*forms of* `def` *carried over from nano-ML* S438b⟩≡ (486a)

```
    VAL    of name * exp
  | VALREC of name * exp
  | EXP    of exp
  | DEFINE of name * (name list * exp)
```

Unit tests are like nano-ML's unit tests, except that the type in a `check-type` or a `check-principal-type` is syntax that has to be translated into a `type_scheme`.

S438c. ⟨*definition of* `unit_test` *for languages with Hindley-Milner types and generated type constructors* S438c⟩≡

```
  datatype unit_test = CHECK_EXPECT      of exp * exp
                     | CHECK_ASSERT      of exp
                     | CHECK_ERROR       of exp
                     | CHECK_TYPE        of exp * tyex
                     | CHECK_PTYPE       of exp * tyex
                     | CHECK_TYPE_ERROR  of def
```

Most of μML's type components are shared with nano-ML or with μHaskell (which didn't make the cut for the book).

S438d. ⟨*Hindley-Milner types with generated type constructors* S438d⟩≡ (S437a)

⟨*foundational definitions for generated type constructors* 485a⟩
⟨*utility functions for generated type constructors* S448d⟩
⟨*representation of Hindley-Milner types* 408⟩
⟨*sets of free type variables in Hindley-Milner types* 433a⟩
⟨*type constructors built into μML and μHaskell* S440d⟩
⟨*types built into μML and μHaskell* S440c⟩
⟨*code to construct and deconstruct function types for μML* S440e⟩
⟨*definition of* `typeString` *for Hindley-Milner types* S431b⟩
⟨*shared utility functions on Hindley-Milner types* S449c⟩
⟨*specialized environments for type schemes* S453c⟩
⟨*extensions that support existential types* S458a⟩

The components of the evaluator and read-eval-print loop are organized as follows:

S438e. ⟨*evaluation, testing, and the read-eval-print loop for μML* S438e⟩≡ (S437a)

⟨*definition of* `namedValueString` *for functional bridge languages* S413a⟩
⟨*definitions of* `match` *and* `Doesn'tMatch` 494b⟩
⟨*definitions of* `eval` *and* `evaldef` *for nano-ML and μML* S429a⟩
⟨*definition of* `processDef` *for μML* S439b⟩
⟨*shared definition of* `withHandlers` S239a⟩
⟨*shared unit-testing utilities* S225a⟩
⟨*definition of* `testIsGood` *for μML* S447a⟩
⟨*shared definition of* `processTests` S226⟩
⟨*shared read-eval-print loop* S237⟩

A basis is a quadruple $\langle \Gamma, \Delta, M, \rho \rangle$. But M is represented implicitly, by the contents of the mutable reference cell `nextIdentity`, so the representation of a basis contains only the components Γ, Δ, and ρ.

S439a. ⟨*definition of* `basis` *for μML* S439a⟩≡ (S437a)

```
type basis = type_env * (ty * kind) env * value env
```

As in other interpreters for statically typed languages, `processDef` first typechecks a definition, then evaluates it. A data definition is handled by function `processDataDef` below. All other definitions are handled by the versions of `typdef` and `evaldef` defined for nano-ML in Chapter 7. The basis for nano-ML lacks a kind environment Δ, so in the formal system, the μML type system uses nano-ML's rules by invoking this rule:

$$\frac{\langle d, \Gamma \rangle \to \langle \Gamma' \rangle}{\langle d, \Gamma, \Delta, M \rangle \to \langle \Gamma', \Delta, M \rangle} \qquad \text{(ReuseDefinition)}$$

S439b. ⟨*definition of* `processDef` *for μML* S439b⟩≡ (S438e)

```
                          processDef : def * basis * interactivity -> basis
```

```
fun processDef (DATA dd, basis, interactivity) =
      processDataDef (dd, basis, interactivity)
  | processDef (d, (Gamma, Delta, rho), interactivity) =
      let val (Gamma', tystring)  = typdef  (d, Gamma)
          val (rho',   valstring) = evaldef (d, rho)
          val _ =
            if echoes interactivity then
              println (valstring ^ " : " ^ tystring)
            else
              ()
      in  (Gamma', Delta, rho')
      end
```

To process a data definition, use `typeDataDef` and `evalDataDef`.

S439c. ⟨*typing and evaluation of* `data` *definitions* S439c⟩≡ (S437a)

```
                 processDataDef : data_def * basis * interactivity -> basis
```

```
fun processDataDef (dd, (Gamma, Delta, rho), interactivity) =
  let val (Gamma', Delta', tystrings) = typeDataDef (dd, Gamma, Delta)
      val (rho', vcons)               = evalDataDef (dd, rho)
      val _ = if echoes interactivity then
                ⟨print the new type and each of its value constructors S439d⟩
              else
                ()
  in  (Gamma', Delta', rho')
  end
```

The name of the new type constructor is printed with its kind, and the name of each value constructor is printed with its type.

S439d. ⟨*print the new type and each of its value constructors* S439d⟩≡ (S439c)

```
let val (T, _, _) = dd
    val (mu, _)   = find (T, Delta')
    val (kind, vcon_types) =
      case tystrings
        of s :: ss => (s, ss)
         | [] => raise InternalError "no kind string"
in  ( println (typeString mu ^ " :: " ^ kind)
    ; ListPair.appEq (fn (K, tau) => println (K ^ " : " ^ tau)) (vcons, vcon_types)
    )
end
```

Everything except a DATA definition is processed by functions typdef and evaldef as defined for nano-ML in Chapter 7. Each of those functions needs an extra case for DATA.

S440a. ⟨*extra case for* typdef *used only in μML* S440a⟩≡ (439a)

```
  | DATA _ => raise InternalError "DATA reached typdef"
```

S440b. ⟨*clause for* evaldef *for datatype definition (μML only)* S440b⟩≡ (S430c)

```
  | evaldef (DATA _, _) = raise InternalError "DATA reached evaldef"
```

S.2 Primitive functions, predefined functions, and the initial basis

S.2.1 Primitive type constructors in μML

In μML, Booleans, lists, pairs, and other algebraic data types are predefined using data definitions. Only four type constructors are defined primitively:

- Integers and symbols, which give types to literal integers and symbols
- Function and argument type constructors, which give types to functions

The first two type constructors are used to make the int and sym types.

S440c. ⟨*types built into μML and μHaskell* S440c⟩≡ (S438d)

```
  val inttype = TYCON inttycon
  val symtype = TYCON symtycon
```

The second two are used to make function types, which we can construct and deconstruct.

S440d. ⟨*type constructors built into μML and μHaskell* S440d⟩≡ (S438d)

```
  val funtycon  = freshTycon "function"
  val argstycon = freshTycon "arguments"
```

Functions asFuntype is the inverse of funtype, satisfying this algebraic law:

$$\texttt{asFuntype} \circ \texttt{funtype} = \texttt{SOME}.$$

S440e. ⟨*code to construct and deconstruct function types for μML* S440e⟩≡ (S438d)

```
funtype   : ty list * ty -> ty
asFuntype : ty -> (ty list * ty) option
```

```
  fun funtype (args, result) =
    CONAPP (TYCON funtycon, [CONAPP (TYCON argstycon, args), result])

  fun asFuntype (CONAPP (TYCON mu, [CONAPP (_, args), result])) =
        if eqTycon (mu, funtycon) then
          SOME (args, result)
        else
          NONE
    | asFuntype _ = NONE
```

Predefined tuple types

Most of the predefined types (bool, list, and unit) are defined in Chapter 8. Only the tuple types are defined here.

S440f. ⟨*predefined μML types* S440f⟩≡ S441a ▷

```
  (data (* * * => *) triple
    [TRIPLE : (forall ['a 'b 'c] ('a 'b 'c -> (triple 'a 'b 'c)))])
```

When defining larger tuples, the notation of the explicit `data` form is untenable. The `implicit-data` form is superior.

S441a. ⟨*predefined μML types* S440f⟩+≡ ◁S440f S470a▷

```
(implicit-data ('a1 'a2 'a3 'a4) 4-tuple
         [T4 of 'a1 'a2 'a3 'a4])
(implicit-data ('a1 'a2 'a3 'a4 'a5) 5-tuple
         [T5 of 'a1 'a2 'a3 'a4 'a5])
(implicit-data ('a1 'a2 'a3 'a4 'a5 'a6) 6-tuple
         [T6 of 'a1 'a2 'a3 'a4 'a5 'a6])
(implicit-data ('a1 'a2 'a3 'a4 'a5 'a6 'a7) 7-tuple
         [T7 of 'a1 'a2 'a3 'a4 'a5 'a6 'a7])
(implicit-data ('a1 'a2 'a3 'a4 'a5 'a6 'a7 'a8) 8-tuple
         [T8 of 'a1 'a2 'a3 'a4 'a5 'a6 'a7 'a8])
(implicit-data ('a1 'a2 'a3 'a4 'a5 'a6 'a7 'a8 'a9) 9-tuple
         [T9 of 'a1 'a2 'a3 'a4 'a5 'a6 'a7 'a8 'a9])
(implicit-data ('a1 'a2 'a3 'a4 'a5 'a6 'a7 'a8 'a9 'a10) 10-tuple
        [T10 of 'a1 'a2 'a3 'a4 'a5 'a6 'a7 'a8 'a9 'a10])
```

S.2.2 Building the initial basis and providing access to predefined types

Other interpreters build an initial basis by starting with an empty basis, adding primitives, and adding predefined functions. But the initial basis for the μML interpreter is built in five stages, not three:

1. Start with an empty basis
2. Add the primitive type constructors `int` and `sym`, producing `primTyconBasis`
3. Add the predefined types, producing `predefinedTypeBasis`

   (At this point, it is possible to implement type inference, which uses the predefined types `list` and `bool` to infer the types of list literals and Boolean literals.)
4. Add the primitives, some of whose types refer to predefined types, producing `primFunBasis`
5. Add the predefined functions, some of whose bodies refer to primitives, producing `initialBasis`

After step 3, the predefined types `list` and `bool` need to be exposed to the type-inference engine, and all the predefined types need to be exposed to the implementations of the primitives. The basis holding the predefined types is called `predefinedTypeBasis`, and the code for the first two steps is implemented here. First, the primitive type constructors:

S441b. ⟨*definitions of* `emptyBasis` *and* `predefinedTypeBasis` S441b⟩≡ (S437a) S442a▷

```
val emptyBasis =
  (emptyTypeEnv, emptyEnv, emptyEnv)
fun addTycon ((t, tycon, kind), (Gamma, Delta, rho)) =
  (Gamma, bind (t, (TYCON tycon, kind), Delta), rho)
val primTyconBasis : basis =
  foldl addTycon emptyBasis (⟨primitive type constructors for μML :: S443c⟩ nil)
```

```
emptyBasis     : basis
primTyconBasis : basis
```

Next, the predefined types. Internal function `process` accepts only `data` definitions, which can be checked without type inference.

S442a. ⟨*definitions of* `emptyBasis` *and* `predefinedTypeBasis` S441b⟩+≡ (S437a) ◁S441b

`predefinedTypeBasis : basis`

```
val predefinedTypeBasis =
  let val predefinedTypes = ⟨predefined μML types, as strings (from ⟨predefined μML types 463a⟩)⟩
      val xdefs = stringsxdefs ("built-in types", predefinedTypes)
      fun process (DEF (DATA dd), b) = processDataDef (dd, b, noninteractive)
        | process _ = raise InternalError "predefined definition is not DATA"
  in  streamFold process primTyconBasis xdefs
  end
```

Internal function `process` is needed because the usual `processDef` isn't yet available. Function `processDef` calls `typdef`; `typdef` invokes type inference; type inference infers `bool` for the type of a condition in `if`; and because `bool` is one of the predefined types, its type constructor isn't known yet. Instead of trying to use `processDef` here, I define an internal function `process`, which processes only data definitions—and that doesn't require type inference.

Once the predefined types have been added to `predefinedTypeBasis`, some of them need to be exposed to the rest of the interpreter. Types `bool`, `list`, `unit`, and `sx` are used to infer types, to write the types of primitive functions, or both. Two more types, `alpha` and `beta`, are used to write the types of polymorphic primitives.

S442b. ⟨*definitions of* `booltype`*,* `listtype`*, and* `unittype` S442b⟩≡ (S437a)

```
local
  val (_, Delta, _) = predefinedTypeBasis
  fun predefined t = fst (find (t, Delta))
  val listtycon = predefined "list"
in
  val booltype     = predefined "bool"
  fun listtype tau = CONAPP (listtycon, [tau])
  val unittype     = predefined "unit"
  val sxtype       = predefined "sx"
  val alpha = TYVAR "'a"
  val beta  = TYVAR "'b"
end
```

To complete the construction of the initial basis, the next step is to add the primitive functions.

S442c. ⟨*implementations of μML primitives and definition of* `initialBasis` S442c⟩≡ (S437a) S443a ▷

```
⟨shared utility functions for building primitives in languages with type inference S421d⟩
⟨utility functions for building nano-ML primitives S422b⟩
val primFunBasis =
  let fun addPrim ((name, prim, tau), (Gamma, Delta, rho)) =
        ( bindtyscheme (name, generalize (tau, freetyvarsGamma Gamma), Gamma)
        , Delta
        , bind (name, PRIMITIVE prim, rho)
        )
  in  foldl addPrim predefinedTypeBasis (⟨primitives for nano-ML and μML :: S470c⟩ nil)
  end
```

And the final step is to add the predefined functions. At this point all of type inference is available, so the construction of the basis can be completed using `readEvalPrintWith`, which calls `processDef`.

S443a. ⟨*implementations of μML primitives and definition of* `initialBasis` S442c⟩+≡ (S437a) ◁S442c S443b▷

```
val initialBasis =
  let val predefinedFuns =
        ⟨predefined μML functions, as strings (from ⟨predefined μML functions 458⟩)⟩
      val xdefs = stringsxdefs ("predefined functions", predefinedFuns)
  in  readEvalPrintWith predefinedFunctionError
                        (xdefs, primFunBasis, noninteractive)
  end
```

Value `primitiveBasis` provides a mockup of the primitive environment. It is used to implement the `-primitives` command-line option defined in Appendix H.

S443b. ⟨*implementations of μML primitives and definition of* `initialBasis` S442c⟩+≡ (S437a) ◁S4

```
val primitiveBasis : basis = (* a mockup, but it's the truth *)
  foldl (fn ((name, prim, tau), (Gamma, Delta, rho)) =>
            (Gamma, Delta, bind (name, PRIMITIVE prim, rho)))
        emptyBasis
        (⟨primitives for nano-ML and μML :: S470c⟩ nil)
val predefs = [] (* not the truth *)
```

S.2.3 Primitive types and functions

Like Typed μScheme, μML has both primitive types and primitive values. Primitive types `int` and `sym` are bound into the kind environment Δ. Other built-in types are either defined in user code, like `list` and `bool`, or they don't have names, like the function type.

S443c. ⟨*primitive type constructors for μML* `::` S443c⟩≡ (S441b)

```
("int", inttycon, TYPE) ::
("sym", symtycon, TYPE) ::
```

μML's primitive values are also nano-ML primitive values, and they are defined in chunk ⟨*primitives for nano-ML and μML* `::` S470c⟩, which is defined in Chapter 7 and Appendix R. The code defined there is reused, but because μML uses `CONVAL` instead of `BOOLV`, `PAIR`, and `NIL`, μML needs new versions of some of the ML functions on which the primitives are built.

The first new function μML needs is the one that defines primitive equality. In μML, polymorphic equality uses the same rules as in full ML; in particular, identical value constructors applied to equal values are considered equal.

S443d. ⟨*utility functions on μML values* S443d⟩≡ S444b▷

```
fun primitiveEquality (v, v') =
  let fun noFun () = raise RuntimeError "compared functions for equality"
  in  case (v, v')
        of (NUM  n1,  NUM  n2)  => (n1 = n2)
         | (SYM  v1,  SYM  v2)  => (v1 = v2)
         | (CONVAL (vcon, vs), CONVAL (vcon', vs')) =>
             vcon = vcon' andalso ListPair.allEq primitiveEquality (vs, vs')
         | (CLOSURE   _, _) => noFun ()
         | (PRIMITIVE _, _) => noFun ()
         | (_, CLOSURE   _) => noFun ()
         | (_, PRIMITIVE _) => noFun ()
         | _ => raise BugInTypeInference
                        ("compared incompatible values " ^ valueString v ^
                         " and " ^ valueString v' ^ " for equality")
  end
val testEquals = primitiveEquality
```

`type basis`	S439a
`bind`	305d
`bindtyscheme`	435c
`BugInType-Inference`	S213d
`CLOSURE`	486d
`CONAPP`	408
`CONVAL`	486d
`DATA`	486a
`DEF`	S214b
`emptyBasis`	S441b
`find`	305b
`freetyvarsGamma`	435d
`fst`	S249b
`generalize`	434b
`InternalError`	S219e
`inttycon`	485d
`noninteractive`	S236c
`NUM`	486d
`predefined-Function-Error`	S215c
`PRIMITIVE`	486d
`primTyconBasis`	S441b
`processDataDef`	S439c
`readEvalPrintWith`	S237
`RuntimeError`	S213b
`streamFold`	S231e
`stringsxdefs`	S233a
`SYM`	486d
`symtycon`	485d
`TYPE`, in μML	S454a
in μML	355a
`TYVAR`	408
`valueString`	S461b

In μML, as in OCaml, comparing functions for equality causes a run-time error. Standard ML has a more elaborate type system which rejects such comparisons during type checking.

The primitive equality used in Molecule (Chapter 9) is almost the same as the version used in μML. The only difference is that in Molecule, a constructed value contains mutable reference cells, not values. Notionally, this version of primitive equality belongs in Appendix T, but because it is so similar to μML's primitive equality, I define it here. That way if I need to change one, I will know also to change the other.

S444a. ⟨*utility functions on μML values* **⟦Molecule⟧** S444a⟩≡ S444c ▷

```
fun primitiveEquality (v, v') =
  let fun noFun () = raise RuntimeError "compared functions for equality"
  in  case (v, v')
        of (NUM  n1,  NUM  n2)  => (n1 = n2)
         | (SYM  v1,  SYM  v2)  => (v1 = v2)
         | (CONVAL (vcon, vs), CONVAL (vcon', vs')) =>
             vcon = vcon' andalso
                    ListPair.allEq primitiveEquality (map ! vs, map ! vs')
         | (CLOSURE   _, _) => noFun ()
         | (PRIMITIVE _, _) => noFun ()
         | (_, CLOSURE   _) => noFun ()
         | (_, PRIMITIVE _) => noFun ()
         | _ => raise BugInTypeInference
                        ("compared incompatible values " ^ valueString v ^
                         " and " ^ valueString v' ^ " for equality")
  end
val testEquals = primitiveEquality
```

The parser for literal S-expressions uses `embedList` to convert a list of S-expressions into an S-expression. The nano-ML version (chunk 308c) uses Standard ML value constructors `PAIR` and `NIL`, but the μML version uses μML value constructors `cons` and `'()`.

S444b. ⟨*utility functions on μML values* S443d⟩+≡ ◁ S443d S444d ▷

```
embedList : value list -> value
```

```
fun embedList []      = CONVAL ("'()", [])
  | embedList (v::vs) = CONVAL ("cons", [v, embedList vs])
```

The version used in Molecule has the same idea, but both value constructors and their arguments are represented differently.

S444c. ⟨*utility functions on μML values* **⟦Molecule⟧** S444a⟩+≡ ◁ S444a S444e ▷

```
fun embedList []      = CONVAL (PNAME "'()", [])
  | embedList (v::vs) = CONVAL (PNAME "cons", [ref v, ref (embedList vs)])
```

The operations that convert between nano-ML Booleans and Standard ML Booleans use nano-ML's `BOOLV`. Again, the μML versions use μML's value constructors.

S444d. ⟨*utility functions on μML values* S443d⟩+≡ ◁ S444b

```
projectBool : value -> bool
embedBool   : bool  -> value
```

```
fun embedBool b =
      CONVAL (if b then "#t" else "#f", [])
fun projectBool (CONVAL ("#t", [])) = true
  | projectBool _                   = false
```

And again the Molecule version is placed here.

S444e. ⟨*utility functions on μML values* **⟦Molecule⟧** S444a⟩+≡ ◁ S444c

```
fun embedBool b = CONVAL (PNAME (if b then "#t" else "#f"), [])
fun projectBool (CONVAL (PNAME "#t", [])) = true
  | projectBool _                         = false
```

S.2.4 Predefined functions

Quite a few predefined functions, including integer comparison and some list functions, appear in Chapter 8. The rest are defined here. Some of the definitions look exactly the same as the corresponding definitions in μScheme or nano-ML.

S445a. ⟨*predefined μML functions* S445a⟩≡ S445b ▷

```
(define and (b c) (if b  c  b))
(define or  (b c) (if b  b  c))
(define not (b)   (if b #f #t))
```

S445b. ⟨*predefined μML functions* S445a⟩+≡ ◁ S445a S445c ▷

```
(define o (f g) (lambda (x) (f (g x))))
(define curry   (f) (lambda (x) (lambda (y) (f x y))))
(define uncurry (f) (lambda (x y) ((f x) y)))
```

S445c. ⟨*predefined μML functions* S445a⟩+≡ ◁ S445b S445d ▷

```
(define caar (xs) (car (car xs)))
(define cadr (xs) (car (cdr xs)))
(define cdar (xs) (cdr (car xs)))
```

The predefined list functions are written using pattern matching. That makes their code simpler than the corresponding functions in nano-ML. Try comparing this code with the code in Section R.2.2 (page S423).

S445d. ⟨*predefined μML functions* S445a⟩+≡ ◁ S445c S445e ▷

```
(define filter (p? xs)
  (case xs
     ['()   '()]
     [(cons y ys)  (if (p? y) (cons y (filter p? ys))
                              (filter p? ys))]))
```

S445e. ⟨*predefined μML functions* S445a⟩+≡ ◁ S445d S445f ▷

```
(define map (f xs)
  (case xs
     ['() '()]
     [(cons y ys) (cons (f y) (map f ys))]))
```

S445f. ⟨*predefined μML functions* S445a⟩+≡ ◁ S445e S445g ▷

```
(define app (f xs)
  (case xs
     ['() UNIT]
     [(cons y ys) (begin (f y) (app f ys))]))
```

S445g. ⟨*predefined μML functions* S445a⟩+≡ ◁ S445f S445h ▷

```
(define reverse (xs) (revapp xs '()))
```

S445h. ⟨*predefined μML functions* S445a⟩+≡ ◁ S445g S445i ▷

```
(define exists? (p? xs)
  (case xs
     ['() #f]
     [(cons y ys) (if (p? y) #t (exists? p? ys))]))
(define all? (p? xs)
  (case xs
     ['() #t]
     [(cons y ys) (if (p? y) (all? p? ys) #f)]))
```

S445i. ⟨*predefined μML functions* S445a⟩+≡ ◁ S445h S446a ▷

```
(define foldr (op zero xs)
  (case xs
     ['() zero]
     [(cons y ys) (op y (foldr op zero ys))]))
```

S446a. ⟨*predefined μML functions* S445a⟩+≡ ◁S445i S446b▷

```
(define foldl (op zero xs)
  (case xs
     ['() zero]
     [(cons y ys) (foldl op (op y zero) ys)]))
```

S446b. ⟨*predefined μML functions* S445a⟩+≡ ◁S446a S446c▷

```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

S446c. ⟨*predefined μML functions* S445a⟩+≡ ◁S446b S446d▷

```
(define max (m n) (if (> m n) m n))
(define min (m n) (if (< m n) m n))
(define negated (n) (- 0 n))
(define mod (m n) (- m (* n (/ m n))))
(define gcd (m n) (if (= n 0) m (gcd n (mod m n))))
(define lcm (m n) (* m (/ n (gcd m n))))
```

S446d. ⟨*predefined μML functions* S445a⟩+≡ ◁S446c S446e▷

```
(define min* (xs) (foldr min (car xs) (cdr xs)))
(define max* (xs) (foldr max (car xs) (cdr xs)))
(define gcd* (xs) (foldr gcd (car xs) (cdr xs)))
(define lcm* (xs) (foldr lcm (car xs) (cdr xs)))
```

S446e. ⟨*predefined μML functions* S445a⟩+≡ ◁S446d S446f▷

```
(define list1 (x)               (cons x '()))
(define list2 (x y)             (cons x (list1 y)))
(define list3 (x y z)           (cons x (list2 y z)))
(define list4 (x y z a)         (cons x (list3 y z a)))
(define list5 (x y z a b)       (cons x (list4 y z a b)))
(define list6 (x y z a b c)     (cons x (list5 y z a b c)))
(define list7 (x y z a b c d)   (cons x (list6 y z a b c d)))
(define list8 (x y z a b c d e) (cons x (list7 y z a b c d e)))
```

S446f. ⟨*predefined μML functions* S445a⟩+≡ ◁S446e S446g▷

```
(define takewhile (p? xs)
  (case xs
     ['() '()]
     [(cons y ys)
        (if (p? y)
            (cons y (takewhile p? ys))
            '())]))
```

S446g. ⟨*predefined μML functions* S445a⟩+≡ ◁S446f

```
(define dropwhile (p? xs)
  (case xs
     ['() '()]
     [(cons y ys)
        (if (p? y)
            (dropwhile p? ys)
            xs)]))
```

S.3 UNIT TESTING

Unit testing is as in nano-ML, except that types in the syntax have to be translated.

S447a. ⟨*definition of* testIsGood *for μML* S447a⟩≡ (S438e) S447b ▷

```
⟨definition of skolemTypes for languages with generated type constructors S448b⟩
⟨shared definitions of typeSchemeIsAscribable and typeSchemeIsEquivalent S427c⟩
fun testIsGood (test, (Gamma, Delta, rho)) =
  let fun ty e = typeof (e, Gamma)
                 handle NotFound x =>
                   raise TypeError ("name " ^ x ^ " is not defined")
      fun ddtystring dd =
        case typeDataDef (dd, Gamma, Delta)
          of (_, _, kind :: _) => kind
           | _ => "???"
      fun deftystring d =
        (case d of DATA dd => ddtystring dd
                 | _ => snd (typdef (d, Gamma)))
        handle NotFound x =>
          raise TypeError ("name " ^ x ^ " is not defined")
      ⟨definitions of check{Expect,Assert,Error{Checks that use type inference S426b⟩
      ⟨definition of checkTypeChecks using type inference S427a⟩

      fun withTranslatedSigma check form (e, sigmax) =
        check (e, txTyScheme (sigmax, Delta))
        handle TypeError msg =>
          failtest ["In (", form, " ", expString e, " ",
                    tyexString sigmax, "), ", msg]
```

A good test has to typecheck.

S447b. ⟨*definition of* testIsGood *for μML* S447a⟩+≡ (S438e) ◁ S447a S448a ▷

```
      val checkTxTypeChecks =
        withTranslatedSigma (checkTypeChecks "check-type") "check-type"
      val checkTxPtypeChecks =
        withTranslatedSigma (checkTypeChecks "check-principal-type")
                            "check-principal-type"

      fun checks (CHECK_EXPECT (e1, e2))    = checkExpectChecks (e1, e2)
        | checks (CHECK_ASSERT e)           = checkAssertChecks e
        | checks (CHECK_ERROR e)            = checkErrorChecks  e
        | checks (CHECK_TYPE  (e, sigmax))  = checkTxTypeChecks (e, sigmax)
        | checks (CHECK_PTYPE (e, sigmax))  = checkTxPtypeChecks (e, sigmax)
        | checks (CHECK_TYPE_ERROR e)       = true

      fun outcome e =
        withHandlers (fn () => OK (eval (e, rho))) () (ERROR o stripAtLoc)
      ⟨asSyntacticValue for μML S448c⟩
      ⟨shared check{Expect,Assert,Error{Passes, which call outcome S224d⟩
      ⟨definitions of check*Type*Passes using type inference S428b⟩
```

And a good test has to pass.

S448a. ⟨*definition of* `testIsGood` *for μML* S447a⟩+≡ (S438e) ◁S447b S457e▷

```
      val checkTxTypePasses =
        withTranslatedSigma checkTypePasses          "check-type"
      val checkTxPtypePasses =
        withTranslatedSigma checkPrincipalTypePasses "check-principal-type"

      fun passes (CHECK_EXPECT (c, e))     = checkExpectPasses    (c, e)
        | passes (CHECK_ASSERT c)          = checkAssertPasses    c
        | passes (CHECK_ERROR c)           = checkErrorPasses     c
        | passes (CHECK_TYPE (c, sigmax))  = checkTxTypePasses    (c, sigmax)
        | passes (CHECK_PTYPE (c, sigmax)) = checkTxPtypePasses   (c, sigmax)
        | passes (CHECK_TYPE_ERROR d)      = checkTypeErrorPasses d

  in  checks test andalso passes test
  end
```

As in Appendix R, `check-principal-type` is implemented by replacing type variables with skolem types, then running the constraint solver. A stream of unique skolem types is defined by using a side-effecting function, which calls `freshTycon` each time a new skolem type is needed.

S448b. ⟨*definition of* `skolemTypes` *for languages with generated type constructors* S448b⟩≡ (S447a)

```
val skolemTypes =
  streamOfEffects (fn () => SOME (TYCON (freshTycon "skolem type")))
```

If a test fails, the syntax of the offending expression is shown, *unless* it's a syntactic value, in which case the value is shown. In μML, a syntactic value is either a literal or a value constructor applied to zero or more syntactic values.

S448c. ⟨`asSyntacticValue` *for μML* S448c⟩≡ (S447b)

`asSyntacticValue : exp -> value option`

```
fun asSyntacticValue (LITERAL v) = SOME v
  | asSyntacticValue (VCONX c)   = SOME (CONVAL (c, []))
  | asSyntacticValue (APPLY (e, es)) =
      (case (asSyntacticValue e, optionList (map asSyntacticValue es))
         of (SOME (CONVAL (c, [])), SOME vs) => SOME (CONVAL (c, vs))
          | _ => NONE)
  | asSyntacticValue _ = NONE
```

S.4 Types and type inference

S.4.1 Support for type equivalence and generativity

As explained in Chapter 8, a type constructor's identity is distinct from its `printName`. Function `eqTycon` uses field `identity`, but messages from the interpreter use function `tyconString`, which returns a type constructor's `printName`.

S448d. ⟨*utility functions for generated type constructors* S448d⟩≡ (S438d) S449a▷

`tyconString : tycon -> string`

```
fun tyconString { identity = _, printName = T } = T
```

To choose the `printName` of a type constructor, I could just use the name in the type constructor's definition. But if a constructor is redefined, you don't want an error message like "cannot make `node` equal to `node`" or "expected `struct point` but argument is of type `struct point`."[1] We can do better. I define a function `freshPrintName` which, when given the name of a type constructor, returns a `printName` that is distinct from prior `printName`s. For example, the first time I de-

[1] The second message is from gcc.

fine node, it prints as node. But the second time I define node, it prints as node@{2}, and so on.

S449a. ⟨*utility functions for generated type constructors* S448d⟩+≡ (S438d) ◁S448d S449b▷

```
                                        freshPrintName : string -> string
local
  val timesDefined : int env ref = ref emptyEnv
                                (* how many times each tycon is defined *)
in
  fun freshPrintName t =
    let val n = find (t, !timesDefined) handle NotFound _ => 0
        val _ = timesDefined := bind (t, n + 1, !timesDefined)
    in  if n = 0 then t  (* first definition *)
        else t ^ "@{" ^ Int.toString (n+1) ^ "}"
    end
end
```

Every type constructor is created by calling function freshTycon, which gives it a fresh printName and a unique identity. Ordinary type constructors have even-numbered identities; odd-numbered identities are reserved for skolem types (Appendix C and Section S.5).

S449b. ⟨*utility functions for generated type constructors* S448d⟩+≡ (S438d) ◁S449a

```
                                              freshTycon : name -> tycon
local
  val nextIdentity = ref 0
  fun freshIdentity () =
    !nextIdentity before nextIdentity := !nextIdentity + 2
in
  fun freshTycon t =
    { identity = freshIdentity(), printName = freshPrintName t }
end
```

S.4.2 Validation of constructor types in data *definitions*

As explained in Section 8.2.3 (page 471), every value constructor in a data definition must have a type that is compatible with the definition. Compatibility is defined formally by inference rules for the type-compatibility judgment $\sigma \preccurlyeq \mu :: \kappa$ (page 488). To implement these rules in a way that doesn't make my head hurt, I define an algebraic data type that shows the four possible shapes of σ, one for each rule. That way I can pattern match on them. The shapes are $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, τ, $\forall \alpha_1, \ldots, \alpha_k . \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, and $\forall \alpha_1, \ldots, \alpha_k . \tau$.

S449c. ⟨*shared utility functions on Hindley-Milner types* S449c⟩≡ (S438d S420a) S449d▷

```
                                                  type scheme_shape
datatype scheme_shape
  = MONO_FUN of              ty list * ty  (* (tau1 ... tauN -> tau) *)
  | MONO_VAL of              ty            (* tau *)
  | POLY_FUN of tyvar list * ty list * ty  (* (forall (a ...) (tau ... -> tau)) *)
  | POLY_VAL of tyvar list * ty            (* (forall (a ...) tau) *)
```

A shape is identified by first looking for a function arrow, then checking to see if the list of α's is empty.

S449d. ⟨*shared utility functions on Hindley-Milner types* S449c⟩+≡ (S438d S420a) ◁S449c

```
                                 schemeShape : type_scheme -> scheme_shape
fun schemeShape (FORALL (alphas, tau)) =
  case asFuntype tau
    of NONE => if null alphas then MONO_VAL tau
               else                POLY_VAL (alphas, tau)
     | SOME (args, result) =>
               if null alphas then MONO_FUN (args, result)
               else                POLY_FUN (alphas, args, result)
```

The type-compatibility judgment can fail in unusually many ways. So my implementation has lots of code for detecting bad outcomes and issuing error messages, and it defines several auxiliary functions:

- Function `appliesMu` says if a type is an application of type constructor μ.
- Function `validateTyvarArguments` ensures that the arguments in a constructor application are distinct type variables; it is defined only on constructor applications.
- Function `validateLengths` checks that the number of type variables in a $\forall$ is the same as the number of type parameters specified by μ's kind.

S450a. ⟨*definition of* `validate`, *for the types of the value constructors of* `T` S450a⟩≡ (489)

```
appliesMu               : ty -> bool
validateTyvarArguments : ty -> unit
validateLengths        : tyvar list * kind list -> unit
```

```
  fun validate (K, sigma as FORALL (alphas, _), mu, kind) =
    let ⟨definition of validateTyvarArguments S451b⟩
        fun appliesMu (CONAPP (tau, _)) = eqType (tau, TYCON mu)
          | appliesMu _ = false
        val desiredType =
          case kind of TYPE    => "type " ^ tyconString mu
                     | ARROW _ => "a type made with " ^ tyconString mu
        fun validateLengths (alphas, argkinds) =
          if length alphas <> length argkinds then
            ⟨for K, complain that alphas is inconsistent with kind S451c⟩
          else
            ()
    in  ⟨validation by case analysis on schemeShape shape and kind S450b⟩
    end
```

The case analysis includes one case per rule. In addition, there is a catchall case that matches when the shape of the type scheme doesn't match the kind of μ.

S450b. ⟨*validation by case analysis on* `schemeShape shape` *and* `kind` S450b⟩≡ (S450a) S451a ▷

```
  case (schemeShape sigma, kind)
    of (MONO_VAL tau, TYPE) =>
         if eqType (tau, TYCON mu) then
           ()
         else
           ⟨type of K should be desiredType but is sigma S451d⟩
     | (MONO_FUN (_, result), TYPE) =>
         if eqType (result, TYCON mu) then
           ()
         else
           ⟨result type of K should be desiredType but is result S451e⟩
     | (POLY_VAL (alphas, tau), ARROW (argkinds, _)) =>
         if appliesMu tau then
           ( validateLengths (alphas, argkinds)
           ; validateTyvarArguments tau
           )
         else
           ⟨type of K should be desiredType but is sigma S451d⟩
```

S451a. ⟨*validation by case analysis on* schemeShape shape *and* kind S450b⟩+≡ (S450a) ◁ S450b

```
   | (POLY_FUN (alphas, _, result), ARROW (argkinds, _)) =>
       if appliesMu result then
         ( validateLengths (alphas, argkinds)
         ; validateTyvarArguments result
         )
       else
         ⟨result type of K should be desiredType but is result S451e⟩
   | _ =>
       ⟨for K, complain that alphas is inconsistent with kind S451c⟩
```

Function validateTyvarArguments, which checks to make sure that the arguments to a constructor application are distinct type variables, is defined as follows:

S451b. ⟨*definition of* validateTyvarArguments S451b⟩≡ (S450a)

```
fun validateTyvarArguments (CONAPP (_, taus)) =
      let fun asTyvar (TYVAR a) = a
            | asTyvar tau =
                raise TypeError ("in type of " ^ K ^ ", type parameter " ^
                                 typeString tau ^ " passed to " ^ T ^
                                 " is not a type variable")
      in  case duplicatename (map asTyvar taus)
            of NONE => ()
             | SOME a =>
                 raise TypeError ("in type of " ^ K ^ ", type parameters " ^
                                  "to " ^ T ^ " must be distinct, but " ^ a ^
                                  " is passed to " ^ T ^ " more than once")
      end
  | validateTyvarArguments (TYCON _) =
      ()  (* happens only when uML is extended with existentials *)
  | validateTyvarArguments _ =
      raise InternalError "impossible type arguments"
```

When validation fails, much of the code that issues error messages is here.

S451c. ⟨*for* K*, complain that* alphas *is inconsistent with* kind S451c⟩≡ (S450–52)

```
(case kind
   of TYPE =>
        raise TypeError ("datatype " ^ T ^ " takes no type parameters, so " ^
                         "value constructor " ^ K ^ " must not be polymorphic")
    | ARROW (kinds, _) =>
          raise TypeError ("datatype constructor " ^ T ^ " expects " ^
                           intString (length kinds) ^ " type parameter" ^
                           (case kinds of [_] => "" | _ => "s") ^
                           ", but value constructor " ^ K ^
                           (if null alphas then " is not polymorphic"
                            else " expects " ^ Int.toString (length alphas) ^
                                 " type parameter" ^
                                 (case alphas of [_] => "" | _ => "s"))))
```

S451d. ⟨*type of* K *should be* desiredType *but is* sigma S451d⟩≡ (S450b S452a)

```
raise TypeError ("value constructor " ^ K ^ " should have " ^ desiredType ^
                 ", but it has type " ^ typeSchemeString sigma)
```

S451e. ⟨*result type of* K *should be* desiredType *but is* result S451e⟩≡ (S450–52)

```
raise TypeError ("value constructor " ^ K ^ " should return " ^ desiredType ^
                 ", but it returns type " ^ typeString result)
```

When μML is extended to include value constructors that have existential types, additional validation is needed.

S452a. ⟨*validation by case analysis on* schemeShape shape *and* kind **[[existentials]]** S452a⟩≡

```
case (schemeShape sigma, kind)
  of (MONO_VAL tau, TYPE) =>
       if eqType (tau, TYCON mu) then
         ()
       else
         ⟨type of K should be desiredType but is sigma S451d⟩
   | (MONO_FUN (_, result), TYPE) =>
       if eqType (result, TYCON mu) then
         ()
       else
         ⟨result type of K should be desiredType but is result S451e⟩
   | (POLY_VAL (alphas, tau), _) =>
       if appliesMu tau orelse eqType (tau, TYCON mu) then
         validateTyvarArguments tau
       else
         ⟨type of K should be desiredType but is sigma S451d⟩
   | (POLY_FUN (alphas, _, result), _) =>
       if appliesMu result orelse eqType (result, TYCON mu) then
         validateTyvarArguments result
       else
         ⟨result type of K should be desiredType but is result S451e⟩
   | _ =>
       ⟨for K, complain that alphas is inconsistent with kind S451c⟩
```

S.4.3 Type predicate for value constructors

During the *evaluation* of a data definition, function isPolymorphicFuntyex is used to determine whether a value constructor is represented as a constructed value or as a function that returns a constructed value (page 490).

S452b. ⟨*utility functions on μML syntax* S452b⟩≡ (S437a)

```
fun isPolymorphicFuntyex (FORALLX (_, tau)) = isPolymorphicFuntyex tau
  | isPolymorphicFuntyex (FUNTYX _)         = true
  | isPolymorphicFuntyex _                  = false
```

S.4.4 Type inference for value constructors

The type of a value constructor is inferred in the same way as the type of a variable: the value constructor's type scheme is instantiated with fresh type variables.

S452c. ⟨*more alternatives for* ty S452c⟩≡ (438c)

```
| ty (VCONX vcon) =
    let val tau =
          freshInstance (findtyscheme (vcon, Gamma))
          handle NotFound _ => raise TypeError ("no value constructor named " ^ vcon)
    in  (tau, TRIVIAL)
    end
```

When a value constructor appears in a pattern, the same technique applies.

S452d. ⟨*definition of function* pvconType S452d⟩≡ (S453a)

```
fun pvconType (K, Gamma) =
  freshInstance (findtyscheme (K, Gamma))
  handle NotFound x => raise TypeError ("no value constructor named " ^ x)
```

The μML interpreter builds on the nano-ML interpreter of Chapter 7, but in Chapter 7 I don't want to reveal the existence of code meant to handle pattern matching. To make the relevant functions available in an acceptable part of the interpreter, I pretend they are part of the `literal` function.

S453a. ⟨*function* `literal`*, to infer the type of a literal constant* S453a⟩≡
  ⟨*definition of function* `pvconType` S452d⟩
  ⟨*definition of function* `pattype` 498⟩
  ⟨*definition of function* `choicetype` 497b⟩

S.4.5 Disjoint union of environments (for pattern matching)

When a pattern match is typechecked, each variable in the pattern induces an environment. Function `disjointUnion` combines the environments and checks for duplicate names, making sure all the variables are distinct. If `disjointUnion` finds a duplicate name, it raises `DisjointUnionFailed`. This exception can be raised only during type inference, not during evaluation.

S453b. ⟨*support for names and environments* S453b⟩≡ (S213a)

```
disjointUnion : 'a env list -> 'a env
```

```
exception DisjointUnionFailed of name
fun disjointUnion envs =
  let val env = List.concat envs
  in  case duplicatename (map fst env)
        of NONE => env
         | SOME x => raise DisjointUnionFailed x
  end
```

S.4.6 Extension of type environments

Because a `type_env` has a special representation, it can't be extended with the `<+>` function from Chapter 5. Instead, I define function `extendTypeEnv`, which takes a `type_env` on the left but a `type_scheme env` on the right.

S453c. ⟨*specialized environments for type schemes* S453c⟩≡ (S438d S420a)

```
extendTypeEnv : type_env * type_scheme env -> type_env
```

```
fun extendTypeEnv (Gamma, bindings) =
  let fun add ((x, sigma), Gamma) = bindtyscheme (x, sigma, Gamma)
  in  foldl add Gamma bindings
  end
```

S.4.7 Elaboration of type syntax

A type expression in μML is just like a type expression in Typed μScheme (page 357). But in Typed μScheme, the name of a type (or type constructor) identifies it completely, and in μML, a type name has to be *translated* into a type constructor (Section 8.7.2, page 489). The translation transforms syntax t (ML type `tyex`) into a type scheme σ (`type_scheme`). The translator also makes sure that the type expression is well kinded.

The relationships between type syntax and the corresponding types are shown in Tables S.1 and S.2. Table S.1 shows the theory and Table S.2 shows the representations used in the interpreter.

Table S.1: Notational correspondence between type syntax and types

Syntax	*Concept*	*Semantics*
t	Type	τ
α	Type variable	α
T	Type name or constructor	μ
$(t_1\ \cdots\ t_n\ \texttt{->}\ t)$	Function type	$\tau_1 \times \cdots \times \tau_n \to \tau$
$(t\ t_1\ \cdots\ t_n)$	Constructor application	$(\tau_1, \ldots, \tau_n)\ \tau$
t	Type scheme	σ
$(\texttt{forall}\ (\alpha_1\ \cdots\ \alpha_n)\ t)$	Quantified type	$\forall \alpha_1, \ldots, \alpha_n . \tau$

Type formation is managed through a kind system, and the kind system for μML is the same as in Typed μScheme, from which the code below is copied.

S454a. ⟨*kinds for typed languages* S454a⟩≡ (S437b S405a) S454b ▷

```
datatype kind = TYPE                          (* kind of all types *)
              | ARROW of kind list * kind     (* kind of many constructors *)
```

S454b. ⟨*kinds for typed languages* S454a⟩+≡ (S437b S405a) ◁ S454a S462d ▷

```
fun eqKind (TYPE, TYPE) = true
  | eqKind (ARROW (args, result), ARROW (args', result')) =
      eqKinds (args, args') andalso eqKind (result, result')
  | eqKind (_, _) = false
and eqKinds (ks, ks') = ListPair.allEq eqKind (ks, ks')
```

The syntax of a type expression is represented as an ML value of type `tyex`.

S454c. ⟨*definition of* `tyex` *for* μ*ML* S454c⟩≡ (S437b)

```
datatype tyex
  = TYNAME  of name                 (* names type or type constructor *)
  | CONAPPX of tyex * tyex list     (* type-level application *)
  | FUNTYX  of tyex list * tyex
  | FORALLX of name list * tyex
  | TYVARX  of name                 (* type variable *)
```

In Typed μScheme, the syntax *is* the type; there's no separate representation. But if you study the representations of `tyex` and `ty` on pages 408 and 486, you might guess what has to be done to convert `tyex` to `ty`:

- Convert function-type syntax to an application of `funty`
- Convert each type name to a `tycon`

The rest of the conversion is structural, plus a little extra work to check that kinds are right. To make the `name`-to-`tycon` conversion easy, and to keep track of kinds, I use a single environment Δ. The environment Δ maps each name both to the type that it stands for and to the kind of that type. The name of a type constructor maps to `TYCON` μ (along with the kind of μ), and the name of a type variable maps to `TYVAR` α (along with the kind of α). The full mapping of `tyex` to `ty` is done by function `txType`.

The type theory that specifies `txType` is a conservative extension of theory of kind checking from Typed μScheme (function `kindof` on page 378). Typed μScheme uses the kinding judgment $\Delta \vdash \tau :: \kappa$, which says that in environment Δ, type τ has kind κ. μML extends that judgment to $\boxed{\Delta \vdash t \rightsquigarrow \tau :: \kappa}$, which says that in environment Δ, type syntax t translates to type τ, which has kind κ. If the types were erased from environment Δ and the syntax $t \rightsquigarrow$ were erased from the judgment $\Delta \vdash t \rightsquigarrow \tau :: \kappa$, the result would be Typed μScheme's kind system. You can prove it for yourself in Exercise 31 from Chapter 8.

Table S.2: Representational correspondence between type syntax and types

Syntax	*Concept*	*Semantics*
`tyex`	Type	`ty`
`TYVARX` α	Type variable	`TYVAR` α
`TYNAME` T	Type name or constructor	`TYCON` μ
`FUNTYEX` $([t_1, \ldots, t_n], t)$	Function type	`funty` $([\tau_1, \ldots, \tau_n], \tau)$
`CONAPPX` $(\tau_1, \ldots, \tau_n)$	Constructor application	`CONAPP` $(\tau, [\tau_1, \ldots, \tau_n])$
`tyex`	Type scheme	`type_scheme`
`FORALLX` $([\alpha_1, \ldots, \alpha_n], t)$	Quantified type	`FORALL` $([\alpha_1, \ldots, \alpha_n], \tau)$

Each clause of `txType` implements the translation rule that corresponds to its syntax. The translation rules, which are shown in Figure 8.6 (page 489), extend Typed μScheme's kinding rules. To start, a type name is looked up in the environment Δ.

$$\frac{T \in \operatorname{dom} \Delta \qquad \Delta(T) = (\tau, \kappa)}{\Delta \vdash T \rightsquigarrow \tau :: \kappa} \qquad (\textsc{KindIntroCon})$$

S455a. ⟨*translation of μML type syntax into types* S455a⟩≡ (S437a) S455b ▷

```
txType : tyex * (ty * kind) env -> ty * kind
```

```
  fun txType (TYNAME t, Delta) =
        (find (t, Delta)
         handle NotFound _ => raise TypeError ("unknown type name " ^ t))
```

A type variable is treated the same way.

$$\frac{\alpha \in \operatorname{dom} \Delta \qquad \Delta(\alpha) = (\tau, \kappa)}{\Delta \vdash \alpha \rightsquigarrow \tau :: \kappa} \qquad (\textsc{KindIntroVar})$$

S455b. ⟨*translation of μML type syntax into types* S455a⟩+≡ (S437a) ◁ S455a S455c ▷

```
    | txType (TYVARX a, Delta) =
        (find (a, Delta)
         handle NotFound _ =>
           raise TypeError ("type variable " ^ a ^ " is not in scope"))
```

Constructor application must be well-kinded.

$$\frac{\begin{array}{c}\Delta \vdash t \rightsquigarrow \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \\ \Delta \vdash t_i \rightsquigarrow \tau_i :: \kappa_i,\ \ 1 \le i \le n\end{array}}{\Delta \vdash (t \ \ t_1 \ \cdots \ t_n) \rightsquigarrow (\tau_1, \ldots, \tau_n)\ \tau :: \kappa} \qquad (\textsc{KindApp})$$

S455c. ⟨*translation of μML type syntax into types* S455a⟩+≡ (S437a) ◁ S455b S456a ▷

```
    | txType (CONAPPX (tx, txs), Delta) =
        let val (tau,  kind)  = txType (tx, Delta)
            val (taus, kinds) =
              ListPair.unzip (map (fn tx => txType (tx, Delta)) txs)
        in  case kind
              of ARROW (argks, resultk) =>
                   if eqKinds (kinds, argks) then
                     (CONAPP (tau, taus), resultk)
                   else
                     ⟨applied type constructor tx has the wrong kind S457b⟩
               | TYPE =>
                     ⟨type tau is not expecting any arguments S457c⟩
        end
```

A function type may be formed only when the argument and result types have kind TYPE.

$$\frac{\Delta \vdash t_i \rightsquigarrow \tau_i :: *,\ \ 1 \le i \le n \qquad \Delta \vdash t \rightsquigarrow \tau :: *}{\Delta \vdash (t_1 \ \cdots \ t_n \ \texttt{->}\ t) \rightsquigarrow \tau_1 \times \cdots \times \tau_n \rightarrow \tau :: *} \qquad \text{(KindFunction)}$$

S456a. ⟨*translation of μML type syntax into types* S455a⟩+≡ (S437a) ◁S455c S456b▷

```
  | txType (FUNTYX (txs, tx), Delta) =
      let val tks = map (fn tx => txType (tx, Delta)) txs
          val tk  = txType (tx, Delta)
          fun notAType (ty, kind) = not (eqKind (kind, TYPE))
          fun thetype  (ty, kind) = ty
      in  if notAType tk then
            raise TypeError ("in result position, " ^
                             typeString (thetype tk) ^ " is not a type")
          else
            case List.find notAType tks
              of SOME tk =>
                   raise TypeError ("in argument position, " ^
                                    typeString (thetype tk) ^ " is not a type")
               | NONE => (funtype (map thetype tks, thetype tk), TYPE)
      end
```

A `forall` quantifier is impermissible in a type—this restriction is what makes the type system a Hindley-Milner type system.

S456b. ⟨*translation of μML type syntax into types* S455a⟩+≡ (S437a) ◁S456a S456c▷

```
  | txType (FORALLX _, _) =
      raise TypeError ("'forall' is permissible only at top level")
```

The elaboration judgment for a type *scheme* is $\boxed{\Delta \vdash t \rightsquigarrow \sigma :: *}$. (Because the kind of a type scheme is always $*$, writing it explicitly is redundant, but it makes the judgment easy to compare with the corresponding judgment for a type.)

In a type *scheme*, `forall` is permitted. Each type variable is given kind $*$.

$$\frac{\begin{array}{c}\alpha_1, \ldots, \alpha_n \text{ are all distinct} \\ \Delta\{\alpha_1 \mapsto (\alpha_1, *), \ldots, \alpha_n \mapsto (\alpha_n, *)\} \vdash t \rightsquigarrow \tau :: *\end{array}}{\Delta \vdash (\texttt{forall}\ (\alpha_1 \ \cdots \ \alpha_n)\ t) \rightsquigarrow \forall \alpha_1, \ldots, \alpha_n . \tau :: *} \qquad \text{(SchemeKindAll)}$$

Variables $\alpha_1, \ldots, \alpha_n$ are guaranteed to be distinct by the parser, so no check is required here.

S456c. ⟨*translation of μML type syntax into types* S455a⟩+≡ (S437a) ◁S456b S457a▷

```
txTyScheme : tyex * (ty * kind) env -> type_scheme
```

```
fun txTyScheme (FORALLX (alphas, tx), Delta) =
      let val Delta' = Delta <+> map (fn a => (a, (TYVAR a, TYPE))) alphas
          val (tau, kind) = txType (tx, Delta')
      in  if eqKind (kind, TYPE) then
            FORALL (alphas, tau)
          else
            raise TypeError ("in " ^ typeSchemeString (FORALL (alphas, tau)) ^
                             ", type " ^ typeString tau ^
                             " has kind " ^ kindString kind)
      end
```

If there's no `forall` in the syntax, a type is also a type scheme (with an empty $\forall$).

$$\frac{\Delta \vdash t \rightsquigarrow \tau :: *}{\Delta \vdash t \rightsquigarrow \forall.\tau :: *} \qquad \textsc{(SchemeKindMonotype)}$$

S457a. ⟨*translation of μML type syntax into types* S455a⟩+≡ (S437a) ◁S456c

```
  | txTyScheme (tx, Delta) =
      case txType (tx, Delta)
        of (tau, TYPE) => FORALL ([], tau)
         | (tau, kind) =>
             raise TypeError ("expected a type, but got type constructor " ^
                              typeString tau ^ " of kind " ^ kindString kind)
```

S.4.8 *Error cases for elaboration of type syntax*

Error messages for bad type syntax are issued here.

S457b. ⟨*applied type constructor* `tx` *has the wrong kind* S457b⟩≡ (S455c)

```
if length argks <> length kinds then
  raise TypeError ("type constructor " ^ typeString tau ^ " is expecting " ^
                   countString argks "argument" ^ ", but got " ^
                   Int.toString (length taus))
else
  let fun findBad n (k::ks) (k'::ks') =
            if eqKind (k, k') then
              findBad (n+1) ks ks'
            else
              raise TypeError ("argument " ^ Int.toString n ^
                               " to type constructor " ^ typeString tau ^
                               " should have kind " ^ kindString k ^
                               ", but it has kind " ^ kindString k')
        | findBad _ _ _ = raise InternalError "undetected length mismatch"
  in  findBad 1 argks kinds
  end
```

S457c. ⟨*type* `tau` *is not expecting any arguments* S457c⟩≡ (S455c)

```
raise TypeError ("type " ^ typeString tau ^ " is not a type constructor, " ^
                 "but it was applied to " ^ countString taus "other type")
```

S457d. ⟨*definition of* `xdef` *(shared)* **⟦assert-types⟧** S457d⟩≡

```
  | ASSERT_PTYPE of name * tyex
```

S457e. ⟨*definition of* `testIsGood` *for μML* S447a⟩+≡ (S438e) ◁S448a

```
fun assertPtype (x, t, (Gamma, Delta, _)) =
  let val sigma_x = findtyscheme (x, Gamma)
      val sigma   = txTyScheme (t, Delta)
      fun fail ss = raise TypeError (concat ss)
  in  if typeSchemeIsEquivalent (VAR x, sigma_x, sigma) then
        ()
      else
        fail ["In (check-principal-type* ", x, " ", typeSchemeString sigma, "),
             , x, " has principal type ", typeSchemeString sigma_x]
  end
```

S.5 Support for existential types

As described in Appendix C, μML can be extended with support for existential types. That extension is implemented here.

First, function asX converts a type scheme to an existential type scheme. For the simple case of one universally quantified and one existentially quantified type variable, the specification looks like this (Appendix C):

$$asX_1(\forall\alpha_1, \beta_1.\tau_1 \to \alpha_1\ \tau) = \forall\alpha_1.(\exists\beta_1.\tau_1) \to \alpha_1\ \tau.$$

As implied by the left-hand side, only a function type can successfully be converted to an existential type.

In general, a converted function type may have any number of universally quantified type variables $\alpha_1, \ldots, \alpha_n$, and likewise any number of existentially quantified type variables $\beta_1, \ldots, \beta_m$. The type is converted by looking at the result type, which must take the form of a type constructor applied to type variables $\alpha_1, \ldots, \alpha_n$. In the converted type, those type variables are universally quantified. Whatever other type variables were quantified in the original `FORALL` are the $\beta_1, \ldots, \beta_m$, and they become existentially quantified.

S458a. ⟨*extensions that support existential types* S458a⟩≡ (S438d) S458b ▷

```
type x_type_scheme
asExistential : type_scheme -> x_type_scheme option
```

```
datatype x_type_scheme
  = FORALL_EXISTS of tyvar list * tyvar list * ty list * ty

fun asExistential (FORALL (alphas_and_betas, tau)) =
  let fun asTyvar (TYVAR a) = a
        | asTyvar _ = raise InternalError "GADT"
      fun typeParameters (CONAPP (mu, alphas)) = map asTyvar alphas
        | typeParameters _ = []
  in  case asFuntype tau
        of SOME (args, result) =>
             let val alphas = typeParameters result
                 val betas = diff (alphas_and_betas, alphas)
             in  SOME (FORALL_EXISTS (alphas, betas, args, result))
             end
         | NONE => NONE
  end
```

An existential type is skolemized with fresh skolem types. A skolem type is represented as a type constructor, but unlike a normal type constructor, it has an odd number as its identity. (If I were starting an implementation from scratch, I would prefer to add `SKOLEM_TYPE` to the representation of `ty`, but I want to reuse the constraint-solving and type-inference code left over from nano-ML, and I prefer a representation that permits me to reuse that code.)

S458b. ⟨*extensions that support existential types* S458a⟩+≡ (S438d) ◁ S458a S460c ▷

```
fun freshSkolem _ =
  let val { identity = id, printName = T } = freshTycon "skolem type"
  in  TYCON { identity = id + 1
            , printName = "skolem type " ^ intString (id div 2)
            }
  end

fun isSkolem { identity = n, printName = _ } = (n mod 2 = 1)
```

Function `freshSkolem` is used to find the type of a value constructor by instantiating the existentially quantified β_i's as fresh skolem types. The judgment $\Gamma \vdash_p K : \tau$ is implemented by a new version of function `pvconType`, which generalizes the earlier version by supporting existential types.

S459a. ⟨*definition of function* `pvconType` **[[existentials]]** S459a⟩≡

```
fun pvconType (K, Gamma) =
  let val sigma = findtyscheme (K, Gamma)
      val sigma' =
        case asExistential sigma
          of NONE => sigma
           | SOME (FORALL_EXISTS (alphas, betas, args, result)) =>
               let val skolems = map freshSkolem betas
                   val theta = tysubst (mkEnv (betas, skolems))
               in  FORALL (alphas, theta (funtype (args, result)))
               end
  in  freshInstance sigma'
  end handle NotFound x => raise TypeError ("no value constructor named " ^ x)
```

Skolem types that are introduced by a pattern match may not appear in the argument type or the result type of that pattern match, as shown by this rule:

$$\frac{C, \Gamma, \Gamma' \vdash p : \tau \qquad C', \Gamma + \Gamma' \vdash e : \tau' \qquad \theta(C \wedge C') \equiv \mathbf{T} \qquad \mathrm{fs}(\theta\Gamma') \cap \mathrm{fs}(\theta\Gamma) = \emptyset \qquad \mathrm{fs}(\theta\Gamma') \cap \mathrm{fs}(\theta(\tau \to \tau')) = \emptyset}{C \wedge C', \Gamma \vdash \texttt{[}p\ \ e\texttt{]} : \tau \to \tau'} \quad (\textsc{ExistentialChoice})$$

The rule is implemented using these representations:

p	e	Γ	Γ'	$\tau \to \tau'$	$C \wedge C'$
`p`	`e`	`Gamma`	`Gamma'`	`ty`	`con`

The free skolem types of $\theta\Gamma'$ are found by looking at all the types bound in $\theta\Gamma'$. But the free skolem types of $\theta\Gamma$ are easier to find; it's enough to look at what θ substitutes for the free type variables of Γ.

S459b. ⟨*check* `p`, `e`, `Gamma'`, `Gamma`, `ty`, *and* `con` *for escaping skolem types* **[[existentials]]** S459b⟩≡

```
let val theta = solve con (* if exn is raised here, we're doomed anyway *)
    val patSkolems =
          typeSchemesFreeSkolems (map snd (typeEnvSubst theta Gamma'))
    val envSkolems =
          typesFreeSkolems (map (varsubst theta) (freetyvarsGamma Gamma))
    val tySkolems  =
          typeFreeSkolems (tysubst theta ty)
    ⟨definitions of skolem functions fail and badType S460b⟩
in  case (inter (patSkolems, tySkolems), inter (patSkolems, envSkolems))
      of (mu :: _,  _) => ⟨fail with skolem escaping into type S459c⟩
       | ([], mu :: _) => ⟨fail with skolem escaping into environment S460a⟩
       | ([], []) => ()
end
```

When $\tau \to \tau'$ has an escaping skolem type, an error message is issued for τ' if possible. If not, one is issued for τ.

S459c. ⟨*fail with skolem escaping into type* S459c⟩≡ (S459b)

```
(case asFuntype (tysubst theta ty)
   of SOME ([tau], tau') =>
        if not (null (inter (patSkolems, typeFreeSkolems tau'))) then
          fail ["right-hand side has ", badType tau']
        else
          fail ["scrutinee is constrained to have ", badType tau]
    | _ => raise InternalError "choice type is not a function type")
```

`asFuntype`	S440e
`badType`	S460b
`con`	497b
`CONAPP`	408
`diff`	S217b
`fail`	S460b
`findtyscheme`	435b
`FORALL`	408
`freetyvarsGamma`	435d
`freshInstance`	434c
`freshTycon`	S449b
`funtype`	S440e
`Gamma`	497b
`Gamma'`	497b
`inter`	S217b
`InternalError`	S219e
`intString`	S214c
`mkEnv`	305e
`NotFound`	305b
`snd`	S249b
`solve`	437b
`type ty`	408
`ty`	497b
`TYCON`	408
`type type_scheme`	408
`typeEnvSubst`	S460e
`TypeError`	S213d
`typeFreeSkolems`	S460d
`typeSchemesFreeSkolems`	S460d
`typesFreeSkolems`	S460d
`tysubst`	410c
`TYVAR`	408
`type tyvar`	408
`varsubst`	410b

If the problem is in the environment, the error message doesn't provide much help.

S460a. ⟨*fail with skolem escaping into environment* S460a⟩≡ (S459b)

```
fail ["skolem type " ^ tyconString mu ^
      " constrains a variable in the environment"]
```

All the failure modes identify the problematic pattern match and raise the `TypeError` exception.

S460b. ⟨*definitions of skolem functions* `fail` *and* `badType` S460b⟩≡ (S459b)

```
fun fail ss =
  raise TypeError (concat (["in choice [", patString p, " ", expString e,
                            "], "] @ ss))
fun badType tau =
  concat ["type ", typeString tau, ", which ",
          (case tau of TYCON _ => "is" | _ => "includes"),
          " an escaping skolem type"]
```

Free skolem types are found by examining every type constructor. To avoid allocating multiple sets of type constructors, I've defined function `addFreeSkolems`, which adds free skolem types to an existing set. This function can be used with `foldl` and an empty set.

S460c. ⟨*extensions that support existential types* S458a⟩+≡ (S438d) ◁S458b S460d▷

```
addFreeSkolems : ty * tycon set -> tycon set
```

```
fun addFreeSkolems (TYCON mu, mus) =
      if isSkolem mu then insert (mu, mus) else mus
  | addFreeSkolems (TYVAR _,  mus) =
      mus
  | addFreeSkolems (CONAPP (tau, taus), mus) =
      foldl addFreeSkolems (addFreeSkolems (tau, mus)) taus
```

Function `addFreeSkolems` can be used to find free skolem types in a type, in a set of types, or in a list of type schemes.

S460d. ⟨*extensions that support existential types* S458a⟩+≡ (S438d) ◁S460c S460e▷

```
typeFreeSkolems  : ty     -> tycon set
typesFreeSkolems : ty set -> tycon set
typeSchemesFreeSkolems : type_scheme list -> tycon set
```

```
fun typeFreeSkolems        tau    = addFreeSkolems (tau, emptyset)
fun typesFreeSkolems       taus   = foldl addFreeSkolems emptyset taus
fun typeSchemesFreeSkolems sigmas =
      typesFreeSkolems (map (fn FORALL (_, tau) => tau) sigmas)
```

The substitution into Γ' is just good enough for use with patterns, where every type scheme in Γ' is a monotype.

S460e. ⟨*extensions that support existential types* S458a⟩+≡ (S438d) ◁S460d

```
typeEnvSubst : subst -> type_scheme env -> type_scheme env
```

```
fun typeEnvSubst theta Gamma' =
  let fun subst (FORALL ([], tau)) = FORALL ([], tysubst theta tau)
        | subst _ = raise InternalError "polytype in pattern"
  in  map (fn (x, sigma) => (x, subst sigma)) Gamma'
  end
```

Finally, vanilla μML, which doesn't support existential types for value constructors, implements the escaping-skolem check by doing nothing.

S460f. ⟨*check* `p`, `e`, `Gamma'`, `Gamma`, `ty`, *and* `con` *for escaping skolem types* S460f⟩≡ (497b)

```
()
```

S.6 Evaluation

For syntactic forms other than the `case` and `data` forms, μML shares its operational semantics and its implementations of `eval` and `ev` with nano-ML. μML adds rules for case expressions, pattern matching, and the `data` definition. All those rules are implemented by code that is shown in Chapter 8. The only case *not* implemented in Chapter 8 is μML's special syntax for a value-constructor expression. Its implementation isn't interesting: like a value variable, a value constructor is evaluated by looking it up in the environment:

S461a. ⟨*more alternatives for* `ev` *for nano-ML and μML* S461a⟩≡ (S429a)

```
| ev (VCONX vcon) = find (vcon, rho)
```

S.7 String conversion

To print a list of values, function `valueString` looks only at the *name* of each value constructor. (When `valueString` is called, the type of the value constructor is no longer available.) This heuristic works so long as `cons` has its expected meaning. If a μML program redefines the `cons` value constructor, chaos may ensue.

S461b. ⟨*definition of* `valueString` *for μML* S461b⟩≡ (S437b) S461c ▷

`valueString : value -> string`

```
fun valueString (CONVAL ("cons", [v, vs])) = consString (v, vs)
  | valueString (CONVAL ("'()",  []))      = "()"
  | valueString (CONVAL (c, []))  = c
  | valueString (CONVAL (c, vs))  =
      "(" ^ c ^ " " ^ spaceSep (map valueString vs) ^ ")"
  | valueString (NUM n      )   =
      String.map (fn #"~" => #"-" | c => c) (Int.toString n)
  | valueString (SYM v      )   = v
  | valueString (CLOSURE   _)   = "<function>"
  | valueString (PRIMITIVE _)   = "<function>"
```

Applications of `cons` get rendered using Scheme syntax for lists.

S461c. ⟨*definition of* `valueString` *for μML* S461b⟩+≡ (S437b) ◁ S461b

```
and consString (v, vs) =
      let fun tail (CONVAL ("cons", [v, vs])) = " " ^ valueString v ^ tail vs
            | tail (CONVAL ("'()", []))       = ")"
            | tail _ =
                raise BugInTypeInference
                  "bad list constructor (or cons/'() redefined)"
      in  "(" ^ valueString v ^ tail vs
          end
```

S461d. ⟨*extra cases of* `expString` *for μML* S461d⟩≡ (S432a)

```
| VCONX vcon => vcon
| CASE (e, matches) =>
    let fun matchString (pat, e) =
          sqbracket (spaceSep [patString pat, expString e])
    in  bracketSpace ("case" :: expString e :: map matchString matches)
    end
```

Patterns are new with μML, but a pattern is rendered in the same way as the corresponding expression.

S462a. ⟨*definition of* `patString` *for μML and μHaskell* S462a⟩≡

```
fun patString WILDCARD    = "_"
  | patString (PVAR x)    = x
  | patString (CONPAT (vcon, []))   = vcon
  | patString (CONPAT (vcon, pats)) =
      "(" ^ spaceSep (vcon :: map patString pats) ^ ")"
```

Rendering for Molecule patterns is also defined here.

S462b. ⟨*definition of* `patString` *for μML and μHaskell*⟦**Molecule**⟧ S462b⟩≡

```
fun patString WILDCARD    = "_"
  | patString (PVAR x)    = x
  | patString (CONPAT (vcon, []))   = vconString vcon
  | patString (CONPAT (vcon, pats)) =
      "(" ^ spaceSep (vconString vcon :: map patString pats) ^ ")"
```

S462c. ⟨*definition of* `tyexString` *for μML* S462c⟩≡ (S437b)

```
fun tyexString (TYNAME t) = t
  | tyexString (CONAPPX (tx, txs)) =
      "(" ^ tyexString tx ^ " " ^ spaceSep (map tyexString txs) ^ ")"
  | tyexString (FORALLX (alphas, tx)) =
      "(forall (" ^ spaceSep alphas ^ ") " ^ tyexString tx ^ ")"
  | tyexString (TYVARX a) = a
  | tyexString (FUNTYX (args, result)) =
      "(" ^ spaceSep (map tyexString args) ^ " -> " ^ tyexString result ^ ")"
```

S462d. ⟨*kinds for typed languages* S454a⟩+≡ (S437b S405a) ◁ S454b

```
fun kindString TYPE = "*"
  | kindString (ARROW (ks, k)) =
      "(" ^ spaceSep (map kindString ks @ ["=>", kindString k]) ^ ")"
```

S.8 Parsing

Where possible, μML's parsers reuse code from other interpreters.

S462e. ⟨*lexical analysis and parsing for μML, providing* `filexdefs` *and* `stringsxdefs` S462e⟩≡ (S437a)

⟨*lexical analysis for μScheme and related languages* S383d⟩
⟨*parsers for single tokens for μScheme-like languages* S385a⟩
⟨*parsers for μML tokens* S463a⟩
⟨*parsers for μML value constructors and value variables* S463c⟩
⟨*parsers and parser builders for formal parameters and bindings* S385c⟩
⟨*parsers and parser builders for Scheme-like syntax* S386d⟩
⟨*parser builders for typed languages* S414c⟩
⟨*parsers for HM types with generated type constructors* S463e⟩
⟨*parsers and* `xdef` *streams for μML* S464a⟩
⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S233a⟩

S.8.1 Identifying μML tokens

From the implementation of μScheme in Appendix O, μML inherits the token parsers `name`, `booltok`, `quote`, and `int`. But unlike μScheme, μML has at least three different species of names: value constructors, value variables, and type variables. To avoid mistakes, I don't use the `name` parser anywhere. A name will always be parsed with a parser that knows what species of name it's looking for.

The name parser is inherited from the implementation of μScheme, so I disable it by rebinding name to a useless value.

S463a. ⟨*parsers for μML tokens* S463a⟩≡ (S462e) S463b ▷

```
val name = () (* don't use me as a parser; too confusing *)
```

The name of a type variable begins with a quote mark.

S463b. ⟨*parsers for μML tokens* S463a⟩+≡ (S462e) ◁ S463a

```
val tyvar =
  quote *> (  curry op ^ "'" <$> namelike
          <?> "type variable (got quote mark)"
           )
```

To identify value constructors and value variables, I define two predicates.

S463c. ⟨*parsers for μML value constructors and value variables* S463c⟩≡ (S462e) S463d ▷

```
fun isVcon x =
  let val lastPart = List.last (String.fields (curry op = #".") x)
      val firstAfterdot = String.sub (lastPart, 0) handle Subscript => #" "
  in  x = "cons" orelse x = "'()" orelse
      Char.isUpper firstAfterdot orelse firstAfterdot = #"#" orelse
      String.isPrefix "make-" x
  end
fun isVvar x = x <> "->" andalso not (isVcon x)
```

Each namelike thing gets its own parser. A value constructor may be not only a suitable name but also a Boolean literal or the empty list.

S463d. ⟨*parsers for μML value constructors and value variables* S463c⟩+≡ (S462e) ◁ S463c

```
val arrow = sat (fn n => n = "->") namelike
val vvar  = sat isVvar namelike
val tyname = vvar
val vcon  =
  let fun isEmptyList (left, right) =
            notCurly left andalso snd left = snd right
      val boolcon = (fn p => if p then "#t" else "#f") <$> booltok
  in  boolcon <|> sat isVcon namelike <|>
      "'()" <$ quote <* sat isEmptyList (pair <$> left <*> right)
  end
```

S.8.2 Parsing types and kinds

Parsers for types and kinds are as in Typed μScheme, except the type parser produces a tyex, not a ty.

S463e. ⟨*parsers for HM types with generated type constructors* S463e⟩≡ (S462e) S463f ▷

```
tyvar : string parser
tyex  : tyex   parser
```

```
fun tyex tokens = (
     TYNAME <$> tyname
 <|> TYVARX <$> tyvar
 <|> usageParsers
        [("(forall (tyvars) type)",
          curry FORALLX <$> bracket ("('a ...)", distinctTyvars) <*> tyex)]
 <|> bracket("(ty ty ... -> ty)",
        arrowsOf CONAPPX FUNTYX <$> many tyex <*>! many (arrow *> many tyex))
) tokens
```

S463f. ⟨*parsers for HM types with generated type constructors* S463e⟩+≡ (S462e) ◁ S463e

```
kind : kind parser
```

```
fun kind tokens = (
      TYPE <$ eqx "*" vvar
  <|> bracket ("arrow kind",
                 curry ARROW <$> many kind <* eqx "=>" vvar <*> kind)
) tokens

val kind = kind <?> "kind"
```

S.8.3 *Parsing patterns*

The distinction between value variable and value constructor is most important in patterns.

S464a. ⟨*parsers and* `xdef` *streams for μML* S464a⟩≡ (S462e) S464b ▷

```
pattern : pat parser
```

```
fun pattern tokens =  (
                WILDCARD    <$  eqx "_" vvar
      <|>       PVAR        <$> vvar
      <|> curry CONPAT      <$> vcon <*> pure []
      <|> bracket ( "(C x1 x2 ...) in pattern"
                  , curry CONPAT <$> vcon <*> many pattern
                  )
       ) tokens
```

S.8.4 *Parsing expressions*

μML's parser is more elaborate then other parsers because it for each binding construct found in nano-ML, it supports two flavors: the standard flavor, which binds variables, and the "patterns everywhere" flavor, which binds patterns. (The case expression, of course, binds only patterns.) To begin, the formal parameters to a function may be variables or patterns. A `vvarFormalsIn` parser takes a string giving the context, because the parser may detect duplicate names. The `patFormals` parser doesn't take the context, because when patterns are used, duplicate names are detected during type checking.

S464b. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁ S464a S464c ▷

```
vvarFormalsIn : string -> name list parser
patFormals    :             pat list parser
```

```
val vvarFormalsIn = formalsOf "(x1 x2 ...)" vvar
val patFormals    = bracket ("(p1 p2 ...)", many pattern)
```

The parser for expressions includes both flavors of bindings, but the "expression builders" used to build the abstract syntax work only with the names flavor. Expression builders that work with the patterns flavor are left as exercises.

S464c. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁ S464b S466b ▷

```
⟨utility functions that help implement μML's syntactic sugar S468e⟩
fun exptable exp =
  let ⟨parsers used in both flavors S464d⟩
      ⟨parsers for bindings to names S465a⟩
      ⟨parsers for bindings to patterns S465b⟩
      ⟨expression builders that expect to bind names S465c⟩
      ⟨μML expression builders that expect to bind patterns S468a⟩
  in  ⟨parsers for expressions that begin with keywords S465d⟩
  end
```

Choices use only patterns, and `letrec` uses only names.

S464d. ⟨*parsers used in both flavors* S464d⟩≡ (S464c)

```
choice   : (pat  * exp)      parser
letrecBs : (name * exp) list parser
```

```
val choice =
  bracket ("[pattern exp]", pair <$> pattern <*> exp)
val letrecBs =
  distinctBsIn (bindingsOf "(x e)" vvar (asLambda "letrec" exp)) "letrec"
```

When syntactic sugar for patterns is introduced, the parsers that might change are `formals`, `letBs`, and `letstarBs`.

S465a. ⟨*parsers for bindings to names* S465a⟩≡ (S464c)

```
val letBs     = distinctBsIn (bindingsOf "(x e)" vvar exp) "let"
val letstarBs = bindingsOf "(x e)" vvar exp
val formals   = vvarFormalsIn "lambda"
```

The syntactic sugar for patterns is recognized by these parsers:

S465b. ⟨*parsers for bindings to patterns* S465b⟩≡ (S464c)

```
val patBs       = bindingsOf "(p e)" pattern exp
val patLetrecBs = map (fn (x, e) => (PVAR x, e)) <$> letrecBs
val patLetBs =
  let fun patVars (WILDCARD)       = []
        | patVars (PVAR x)         = [x]
        | patVars (CONPAT (_, ps)) = List.concat (map patVars ps)
      fun check (loc, bs) =
        let val xs = List.concat (map (patVars o fst) bs)
        in  nodups ("bound name", "let") (loc, xs) >>=+ (fn _ => bs)
        end
  in  check <$>! @@ patBs
  end
val patFormals = patFormals (* defined above *)
```

When syntactic sugar for patterns is introduced, it will be necessary to define new versions of the expression-builders `lambda`, `lambdastar`, and `letx`.

S465c. ⟨*expression builders that expect to bind names* S465c⟩≡ (S464c)

```
letx       : let_flavor -> (name * exp) list -> exp -> exp
lambda     : name list -> exp -> exp
lambdastar : (pat list * exp) list -> exp error
```

```
fun letx letkind bs e = LETX (letkind, bs, e)
fun lambda xs e = LAMBDA (xs, e)
fun lambdastar clauses = ERROR "lambda* is left as an exercise"
```

Once the `lambdastar` exercise is completed, the parser below should recognize all the expressions. It also recognizes `while` and `set`, which it rejects with an error message.

S465d. ⟨*parsers for expressions that begin with keywords* S465d⟩≡ (S464c)

```
usageParsers
  [ ("(if e1 e2 e3)",            curry3 IFX    <$> exp  <*> exp <*> exp)
  , ("(begin e1 ...)",                  BEGIN  <$> many exp)
  , ("(lambda (names) body)",           lambda <$> formals <*> exp)
  , ("(lambda* (pats) exp ...)",
       lambdastar <$>!
       many1 (bracket
                ( "[(pat ...) e]"
                , pair <$> (bracket ("(pat ...)", many pattern)) <*> exp
                )))
  , ("(let (bindings) body)",     letx   LET     <$> letBs     <*> exp)
  , ("(letrec (bindings) body)",  letx   LETREC  <$> letrecBs  <*> exp)
  , ("(let* (bindings) body)",    letx   LETSTAR <$> letstarBs <*> exp)
  , ("(case exp [pattern exp] ...)", curry CASE <$> exp <*> many choice)

  , ("(while e1 e2)", exp  *> exp <!>
                              "uML does not include 'while' expressions")
  , ("(set x e)",     vvar *> exp <!>
                              "uML does not include 'set' expressions")
  ⟨rows added to μML's exptable in exercises S466a⟩
  ]
```

In the exercises, you'll add more forms of `exp`.

S466a. ⟨*rows added to μML's* `exptable` *in exercises* S466a⟩≡ (S465d)

```
(* you add this bit *)
```

With the keyword expressions parsed by `exptable`, parsers for atomic expressions and full expressions follow.

S466b. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁S464c S466c▷

```
atomicExp : exp parser
exp       : exp parser
```

```
val atomicExp =  VAR              <$> vvar
             <|> VCONX            <$> vcon
             <|> (LITERAL o NUM)  <$> int

fun exp tokens = (
     atomicExp
 <|> quote *> (LITERAL <$> sexp)
 <|> exptable exp
 <|> leftCurly <!> "curly brackets are not supported"
 <|> left *> right <!> "empty application"
 <|> bracket ("function application", curry APPLY <$> exp <*> many exp)
) tokens
```

S.8.5 *Parsing definitions*

The parser for `implicit-data` is defined separately. The syntax is passed to function `makeExplicit` (page S469), which desugars it into what `DATA` expects.

S466c. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁S466b S467c▷

```
makeExplicit : implicit_data_def -> data_def
implicitData : def parser
```

```
⟨definition of makeExplicit, to translate implicit-data to data S469d⟩
val tyvarlist = bracket ("('a ...)", many1 tyvar)
val optionalTyvars = (fn alphas => getOpt (alphas, [])) <$> optional tyvarlist
val implicitData =
  let fun vc c taus = IMPLICIT_VCON (c, taus)
      val vconDef =  vc <$> vcon <*> pure []
                 <|> bracket ("(vcon of ty ...)",
                              vc <$> vcon <* eqx "of" vvar <*> many1 tyex)
  in  usageParsers
      [("(implicit-data [('a ...)] t vcon ... (vcon of ty ...) ...)"
       , (DATA o makeExplicit) <$>
         (curry3 IMPLICIT_DATA <$> optionalTyvars <*> tyname <*> many vconDef)
       )]
  end
```

Definitions are parsed using a small collection of internal functions.

S466d. ⟨*parser for binding to names* S466d⟩≡ (S467c)

```
formals : name list parser
```

```
val formals = vvarFormalsIn "define"
```

S466e. ⟨*parsers for clausal definitions, a.k.a.* `define*` S466e⟩≡ (S467c)

```
lhs    : (name * pat list) parser
clause : (name * (pat list * exp)) parser
```

```
val lhs =
  bracket ("(f p1 p2 ...)", pair <$> vvar <*> many pattern)
val clause =
  bracket ("[(f p1 p2 ...) e]",
           (fn (f, ps) => fn e => (f, (ps, e))) <$> lhs <*> exp)
```

S467a. ⟨*definition builders used in all parsers* S467a⟩≡ (S467c)

```
Kty  : (vcon * tyex) parser
data : kind -> name -> (vcon * tyex) list -> def
```

```
val Kty = typedFormalOf vcon (kw ":") tyex
fun data kind name vcons = DATA (name, kind, vcons)
```

S467b. ⟨*definition builders that expect to bind names* S467b⟩≡ (S467c)

```
fun define f xs body = DEFINE (f, (xs, body))
fun definestar _ = ERROR "define* is left as an exercise"
```

True definitions are parsed by `def`.

S467c. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁S466c S467d▷

```
def : def parser
```

```
val def =
  let ⟨parser for binding to names S466d⟩
      ⟨parsers for clausal definitions, a.k.a. define* S466e⟩
      ⟨definition builders that expect to bind names S467b⟩
      ⟨definition builders used in all parsers S467a⟩
      ⟨μML definition builders that expect to bind patterns (from chunk 697b)⟩
  in  usageParsers
      [ ("(define f (args) body)",    define       <$> vvar <*> formals <*> exp)
      , ("(define* (f pats) e ...)",  definestar   <$>! many1 clause)
      , ("(val x e)",                 curry VAL    <$> vvar <*> exp)
      , ("(val-rec x e)",             curry VALREC <$> vvar <*> asLambda "val-rec" exp)
      , ("(data kind t [vcon : type] ...)",
                                      data <$> kind <*> tyname <*> many Kty)
      ]
  end
```

Unit tests are parsed by `testtable`.

S467d. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁S467c S467e▷

```
testtable : unit_test parser
```

```
val testtable = usageParsers
  [ ("(check-expect e1 e2)",          curry CHECK_EXPECT     <$> exp <*> exp)
  , ("(check-assert e)",                    CHECK_ASSERT     <$> exp)
  , ("(check-error e)",                     CHECK_ERROR      <$> exp)
  , ("(check-type e tau)",            curry CHECK_TYPE       <$> exp <*> tyex)
  , ("(check-principal-type e tau)",  curry CHECK_PTYPE      <$> exp <*> tyex)
  , ("(check-type-error e)",                CHECK_TYPE_ERROR <$> (def <|> implicitData
                                                                 <|> EXP <$> exp))

  ]
```

The other extended definitions are parsed by `xdeftable`.

S467e. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁S467d S467h▷

```
xdeftable : xdef parser
```

```
val xdeftable = usageParsers
  [ ("(use filename)", USE <$> namelike)
  ⟨rows added to μML's xdeftable in exercises S467f⟩
  ]
```

S467f. ⟨*rows added to μML's* `xdeftable` *in exercises* S467f⟩≡ (S467e)

```
(* you add this bit *)
```

S467g. ⟨*rows added to μML's* `xdeftable` *in exercises* **[[assert-types]]** S467g⟩≡

And as usual, all the definitions together are parsed by `xdef`.

S467h. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁S467e S470b▷

```
xdef : xdef parser
```

```
val xdef  =  TEST <$> testtable
         <|>          xdeftable
         <|> DEF  <$> (def <|> implicitData)
         <|> badRight "unexpected right bracket"
         <|> DEF <$> EXP <$> exp
         <?> "definition"
val xdefstream = interactiveParsedStream (schemeToken, xdef)
```

S.9 Support for syntactic-sugar exercises

μML can support “patterns everywhere” through syntactic sugar, which is left for you to add (exercises, Chapter 8). You just have to fill in some code chunks with appropriate parsers:

S468a. ⟨*μML expression builders that expect to bind patterns* S468a⟩≡ (S464c)

```
(* you can redefine letx, lambda, and lambdastar here *)
```

S468b. ⟨*μML definition builders that expect to bind patterns* **[[prototype]]** S468b⟩≡

```
(* you can redefine 'define' and 'definestar' here *)
```

S468c. ⟨*rows added to μML’s* `xdeftable` *in exercises* **[[prototype]]** S468c⟩≡

```
(* you can add a row for 'val' here *)
```

To desugar the new syntax, you will sometimes need to find a variable that is not free in a given expression. If you have done Exercise 10 in Chapter 5 (page 325), you’re almost there. Use that code to complete function `freeIn` here.

S468d. ⟨*utility functions that help implement μML’s syntactic sugar* **[[prototype]]** S468d⟩≡

```
freeIn : exp -> name -> bool
```

```
fun freeIn exp y =
  let fun has_y (CASE (e, choices)) =
            has_y e orelse (List.exists choice_has_y) choices
        | has_y _ =
            raise LeftAsExercise "free variable of an expression"
      and choice_has_y (p, e) = not (pat_has_y p) andalso has_y e
      and pat_has_y (PVAR x) = x = y
        | pat_has_y (CONPAT (_, ps)) = List.exists pat_has_y ps
        | pat_has_y WILDCARD = false
  in  has_y exp
  end
```

Once `freeIn` is implemented, you can come up with fresh variables by using the helper functions below. Function `freshVar` returns a variable that is not free in a given expression. The supply of variables is infinite, so the exception should never be raised.

S468e. ⟨*utility functions that help implement μML’s syntactic sugar* S468e⟩≡ (S464c) S468f ▷

```
varsupply : name stream
freshVar  : exp -> name
```

```
val varsupply =
  streamMap (fn n => "x" ^ intString n) naturals
fun freshVar e =
  case streamGet (streamFilter (not o freeIn e) varsupply)
    of SOME (x, _) => x
     | NONE => raise InternalError "unable to create a fresh variable"
```

Function `freshVars` returns as many fresh variables as there are elements in `xs`.

S468f. ⟨*utility functions that help implement μML’s syntactic sugar* S468e⟩+≡ (S464c) ◁ S468e S468g ▷

```
freshVars : exp -> 'a list -> name list
```

```
fun freshVars e xs =
  streamTake (length xs, streamFilter (not o freeIn e) varsupply)
```

In the concrete syntax of `lambda`, `lambda*`, and `define*`, a sequence of names stands for a tuple, and similarly a sequence of patterns stands for a tuple pattern. Sequences are turned into expressions or patterns, respectively, by functions `tupleexp` and `tuplepat`. Each tuple expression or pattern has to use an appropriate value constructor as defined on page S440. That value constructor is chosen by calling `tupleVcon` with the list of names or patterns involved.

S468g. ⟨*utility functions that help implement μML’s syntactic sugar* S468e⟩+≡ (S464c) ◁ S468f S469a ▷

```
tupleexp  : name list -> exp
tuplepat  : pat  list -> pat
tupleVcon : 'a   list -> vcon
```

```
fun tupleVcon xs = case length xs
                     of 2 => "PAIR"
                      | 3 => "TRIPLE"
                      | n => "T" ^ intString n
```

S469a. ⟨*utility functions that help implement μML's syntactic sugar* S468e⟩+≡ (S464c) ◁ S468g S469b ▷

```
fun tupleexp [x] = VAR x
  | tupleexp xs  = APPLY (VCONX (tupleVcon xs), map VAR xs)

fun tuplepat [x] = x
  | tuplepat xs  = CONPAT (tupleVcon xs, xs)
```

The free variables in a pattern can be found by function freePatVars.

S469b. ⟨*utility functions that help implement μML's syntactic sugar* S468e⟩+≡ (S464c) ◁ S469a

```
freePatVars : pat -> name set
```

```
fun freePatVars (PVAR x)         = insert (x, emptyset)
  | freePatVars (WILDCARD)       = emptyset
  | freePatVars (CONPAT (_, ps)) = foldl union emptyset (map freePatVars ps)
```

S.10 Syntactic sugar for implicit-data

An implicit data definition gives type parameters, the name of the type constructor, and definitions for one or more value constructors.

S469c. ⟨*definition of* implicit_data_def *for μML* S469c⟩≡ (S437b)

```
datatype implicit_data_def
  = IMPLICIT_DATA of tyvar list * name * implicit_vcon list
and implicit_vcon
  = IMPLICIT_VCON of vcon * tyex list
```

An implicit data definition is translated into an explicit data definition by function makeExplicit. In this translation, as long as all the t's elaborate to σ's, each σ satisfies the compatibility judgment $\sigma \preccurlyeq \mu :: \kappa$.

S469d. ⟨*definition of* makeExplicit*, to translate* implicit-data *to* data S469d⟩≡ (S466c)

```
makeExplicit : implicit_data_def -> data_def
```

```
fun makeExplicit (IMPLICIT_DATA ([], t, vcons)) =
      let val tx = TYNAME t
          fun convertVcon (IMPLICIT_VCON (K, []))  = (K, tx)
            | convertVcon (IMPLICIT_VCON (K, txs)) = (K, FUNTYX (txs, tx))
      in  (t, TYPE, map convertVcon vcons)
      end
  | makeExplicit (IMPLICIT_DATA (alphas, t, vcons)) =
      let val kind = ARROW (map (fn _ => TYPE) alphas, TYPE)
          val tx   = CONAPPX (TYNAME t, map TYVARX alphas)
          fun close tau = FORALLX (alphas, tau)
          fun vconType (vcon, [])  = tx
            | vconType (vcon, txs) = FUNTYX (txs, tx)
          fun convertVcon (IMPLICIT_VCON (K, []))  =
                                   (K, close tx)
            | convertVcon (IMPLICIT_VCON (K, txs)) =
                                   (K, close (FUNTYX (txs, tx)))
    in  (t, kind, map convertVcon vcons)
    end
```

S.11 S-expression reader

To enable everyone to write more interesting μML programs, I define a representation of S-expressions in μML and a primitive that reads S-expressions from a file. I had hoped to extend every single bridge language with a little library for reading data from files, but there wasn't quite time. Shipping is also a feature.

An S-expression is a Boolean, symbol, number, or list of S-expressions.

S470a. ⟨*predefined μML types* S440f⟩+≡ ◁S441a

```
(data * sx
   [Sx.B : (bool -> sx)]
   [Sx.S : (sym  -> sx)]
   [Sx.N : (int  -> sx)]
   [Sx.L : ((list sx)  -> sx)])
```

These S-expressions are read by a little parser.

S470b. ⟨*parsers and* `xdef` *streams for μML* S464a⟩+≡ (S462e) ◁S467h

```
sxstream : string * line stream * prompts -> value stream
```

```
local
  fun sxb b = CONVAL ("Sx.B", [embedBool b])
  fun sxs s = CONVAL ("Sx.S", [SYM s])
  fun sxn n = CONVAL ("Sx.N", [NUM n])
  fun sxlist sxs = CONVAL("Sx.L", [embedList sxs])

  fun sexp tokens = (
         sxb <$> booltok
     <|> sxs <$> (notDot <$>! @@ namelike)
     <|> sxn <$> int
     <|> leftCurly <!> "curly brackets may not be used in S-expressions"
     <|> (fn v => sxlist [sxs "quote", v]) <$> (quote *> sexp)
     <|> sxlist <$> bracket ("list of S-expressions", many sexp)
    ) tokens
  val sexp = sexp <?> "S-expression"
in
  val sxstream = interactiveParsedStream (schemeToken, sexp)
end
```

The `read` primitive uses the parser to build a list containing the S-expressions read from a file.

S470c. ⟨*primitives for nano-ML and μML* `::` S470c⟩≡ (S442c S443b S425c)

```
("read", unaryOp (fn (SYM s) =>
                       let val fd = TextIO.openIn s
                             handle _ =>
                               raise RuntimeError ("Cannot read file " ^ s)
                           val sxs = sxstream (s, filelines fd, noPrompts)
                       in  embedList (listOfStream sxs)
                           before TextIO.closeIn fd
                       end
                     | _ => raise BugInTypeInference "read got non-symbol")
         , funtype ([symtype], listtype sxtype)) ::
```

S.12 Useful μML functions

Many of the examples in Chapter 8 produce data that is sophisticated enough to warrant help manipulating it. Below are a higher-order printing library and a library for drawing graphs with `dot`, the Graphviz tool.

S.12.1 Coding and printing strings as functions

Because μML doesn't have strings, printing complicated things is a pain. But wait! A string s can be coded as a function of no arguments that, when called, prints s. To help distinguish such coded strings from other functions, I put the function in a record of type `printable`. A value of type `printable` represents a thing that can be printed, and such values can be made and combined using these functions:

S471a. ⟨*printers.uml* S471a⟩≡ S471b ▷

```
(record printable ([print : ( -> unit)]))
(check-type print>>     [printable -> unit])
(check-type println>>   [printable -> unit])

(check-type >>val       [forall ('a) ('a -> printable)])
(check-type >>vals      [forall ('a) ((list 'a) -> printable)])
(check-type >>char      [int -> printable])
(check-type ^           [printable printable -> printable]) ; concatenate
(check-type >>concat    [(list printable) -> printable])    ; concatenate
(check-type >>space-sep [(list printable) -> printable])    ; concat w/spaces
(check-type >>comma-sep [(list printable) -> printable])    ; concat w/commas
(check-type >>newline   printable)
(check-type >>space     printable)
(check-type >>parens    [printable -> printable])  ; wrap in brackets
(check-type >>wrap      [int int -> (printable -> printable)])
                                        ; wrap in any two characters given
```

The implementations use `lambda` as described in Chapter 2.

S471b. ⟨*printers.uml* S471a⟩+≡ ◁ S471a S471c ▷

```
(define print>>   (p) ((printable-print p)))
(define println>> (p) (begin (print>> p) (printu 10) UNIT))
(define >>char (u)    (make-printable (lambda () (printu u))))
(define >>val  (v)    (make-printable (lambda () (print  v))))
(define >>concat (ps) (make-printable (lambda () (app print>> ps))))
(define ^ (p1 p2)
  (make-printable (lambda () (begin (print>> p1) (print>> p2)))))
```

S471c. ⟨*printers.uml* S471a⟩+≡ ◁ S471b S472a ▷

```
(define >>sep (sep) ; return list printer with given separator
 (letrec
  ((p (lambda (xs)
        (case xs
           [(cons y '()) (print>> y)]
           [(cons y ys)  (begin (print>> y) (print>> sep) (p ys))]
           ['() UNIT]))))
  (lambda (xs) (make-printable (lambda () (p xs))))))
```

S472a. ⟨*printers.uml* S471a⟩+≡ ◁S471c

```
(val >>space   (>>char 32))
(val >>newline (>>char 10))
(val >>comma   (>>char 44))
(val >>space-sep (>>sep >>space))
(define ^space (p1 p2) (^ p1 (^ >>space p2)))
(val >>vals    (lambda (xs) (>>space-sep (map >>val xs))))
(val >>comma-sep (>>sep (^ >>comma >>space)))
(define >>wrap (open close)
  (lambda (p) (>>concat (list3 (>>char open) p (>>char close)))))
(val >>parens (>>wrap 40 41))
```

S.12.2 Drawing simple figures in PostScript

A couple of the figures in Chapter 8 are made by running μML code to produce PostScript. In case you want to make similar figures, I provide my μML code for making circles, disks, and lines in PostScript.

S472b. ⟨*postscript.uml* S472b⟩≡ S472c▷

```
ps-draw-circle : (int int int -> unit)
ps-draw-disk   : (int int int -> unit)
```

```
(use printers.uml)

(define ps-draw-circle (x y radius)
 (let* ([line (>>space-sep (list2 (>>vals (list5 x y radius 0 360))
                                  (>>vals '(arc closepath stroke))))])
    (println>> line)))
(define ps-draw-disk (x y radius)
 (let* ([disk (>>space-sep
                 (list2 (>>vals (list5 x y radius 0 360))
                        (>>vals '(arc closepath 0.0 setgray fill))))])
    (println>> disk)))
```

S472c. ⟨*postscript.uml* S472b⟩+≡ ◁S472b S472d▷

```
(val ps-first-line '%!PS-Adobe-1.0)
```

Function `ps-draw-polyline` has an annoyingly polymorphic principal type. To clarify its intended use, I show a less general type using `check-type`.

S472d. ⟨*postscript.uml* S472b⟩+≡ ◁S472c

```
(check-type ps-draw-polyline
     [forall ('a) (sym ('a -> int) ('a -> int) (list 'a) -> unit)])
(define ps-draw-polyline (width x-of y-of pts)
  (let* ([setwidth (>>vals (list2 width 'setlinewidth))]
         [first    (car pts)]
         [rest     (cdr pts)]
         [point    (lambda (p) (>>vals (list2 (x-of p) (y-of p))))]
         [move     (lambda (p) (^space (point p) (>>val 'moveto)))]
         [draw     (lambda (p) (^space (point p) (>>val 'lineto)))]
         [finish   (>>vals '(0.0 setgray stroke))]
         [line (>>space-sep (list5 setwidth
                                   (>>val 'newpath)
                                   (move first)
                                   (>>space-sep (map draw rest))
                                   finish))])
     (println>> line)))
```

Appendix T contents

T

Supporting code for the Molecule interpreter

Alone among the bridge languages, Molecule is implemented entirely in an appendix: Chapter 9 shows no code. And the implementation is not nearly as thoroughly explained as the other implementations. If you've gotten this far, you know what abstract syntax looks like, you've seen lexers and parsers, and most importantly, you know to turn inference rules into evaluators and type checkers.

Molecule's implementation takes up almost this whole appendix. But at the very end of the appendix, I present an implementation of the `HISTOGRAM` interface, as promised in Chapter 9.

T.1 Organizing code chunks into an interpreter

Unlike the other interpreters, the Molecule interpreter includes a function `unimp`. That function is called in a few corner cases where there was something that either I didn't see how to implement or (more likely) ran out of time to implement.

S475. ⟨*mcl.sml* S475⟩≡

```
exception Unimp of string (* raised if a feature is not implemented *)
fun unimp s = raise Unimp s
fun concatMap f = List.concat o map f  (* List.concatMap is not in Moscow ML *)
⟨exceptions used in languages with type checking S213c⟩
⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization S213a⟩
⟨prettyprinting combinators S526d⟩

⟨abstract syntax and values for Molecule S478c⟩
⟨support for operator overloading in Molecule S517e⟩
⟨lexical analysis and parsing for Molecule, providing filexdefs and stringsxdefs S532b⟩

⟨environments for Molecule's defined names S496a⟩

⟨type checking for Molecule S496c⟩
⟨substitutions for Molecule S513c⟩
⟨elaboration of Molecule type syntax into types S515c⟩

⟨evaluation, testing, and the read-eval-print loop for Molecule S518b⟩

⟨functions for building primitives when types are checked S393a⟩
⟨list of (typed) primitives for each primitive module S488d⟩
⟨primitive modules and definition of initialBasis S492a⟩
⟨function runStream, which evaluates input given initialBasis S240b⟩
⟨look at command-line arguments, then run S240c⟩
```

A "module constructor" names a module. Just like a tycon in μML, it's generative. A module constructor is generated for each definition of a named module, and also for each formal parameter to a module function.

"Module identifier" is either a modcon or is the special identifier `NAMEDMODTY` or `MODTYPLACEHOLDER`, which is attached to components in named module types.

T.2.1 Module identifiers and paths

A module may be identified by its module constructor (`modcon`, similar to `tycon` in μML), or it may just be a placeholder. (A path in a module-type definition starts with `MODTYPLACEHOLDER`.) A fresh module constructor can be generated by function `genmodident`.

S476a. ⟨*paths for Molecule* S476a⟩≡ (S478c) S476b ▷

`genmodident : name -> modident`

```
type modcon = { printName : name, serial : int }
datatype modident = MODCON of modcon | MODTYPLACEHOLDER of name
```

⟨*definition of function* `genmodident` S476c⟩

Paths are formed from names using dot notation and functor application. Type `pathex` is the syntax, and `path` is the elaborated form.

S476b. ⟨*paths for Molecule* S476a⟩+≡ (S478c) ◁ S476a

```
datatype 'modname path' = PNAME of 'modname
                        | PDOT of 'modname path' * name
                        | PAPPLY of 'modname path' * 'modname path' list

type pathex = name located path'
type path   = modident path'
```

A module identifier is genreated as follows

S476c. ⟨*definition of function* `genmodident` S476c⟩≡ (S476a)

```
local
  val timesDefined : int env ref = ref emptyEnv
     (* how many times each modident is defined *)
in
  fun genmodident name =
    let val n = find (name, !timesDefined) handle NotFound _ => 0
        val n = 0  (* grotesque hack *)
        val _ = timesDefined := bind (name, n + 1, !timesDefined)
    in  MODCON { printName = name, serial = n }
    end
end
```

The "grotesque hack" is there for two reasons:

- Arrays aren't handled in the right way—`Array` and `ArrayCore` are conflated, with the dreadful result that the module identifier for `Array` is generated twice, but those modules need to be the same.
- The serial-number printing is too aggressive. Printing of a module identifier should depend on the environment in which the module identifier is used.

The hack puts a huge loophole in the type system—you can define two different modules with the same name, and the system will assignm them the same identifier. Broken, broken, broken.

T.2.2 Types and type equality

There are no type constructors or type applications. (Instead, there are module constructors and module applications, which of course are found in type path.) Every type is either a component of a module or a function type. As with modules, tyex is the syntax and ty is the elaborated form.

S477a. ⟨*definition of* ty *for Molecule* S477a⟩≡ (S478c)

```
datatype 'modname ty' = TYNAME of 'modname path'
                      | FUNTY  of 'modname ty' list * 'modname ty'
                      | ANYTYPE   (* type of (error ...) *)
type tyex = name located ty'
type ty   = modident ty'
```

Type ANYTYPE is equal to any type, and otherwise type equality is structural. (It really hinges on *path* equality.)

S477b. ⟨*type equality for Molecule* S477b⟩≡ (S496c)

```
eqType  : ty      * ty      -> bool
eqTypes : ty list * ty list -> bool
```

```
fun eqType (TYNAME p, TYNAME p') = p = p'
  | eqType (FUNTY (args, res), FUNTY (args', res')) =
      eqTypes (args, args') andalso eqType (res, res')
  | eqType (ANYTYPE, _) = true
  | eqType (_, ANYTYPE) = true
  | eqType _ = false
and eqTypes (taus, tau's) = ListPair.allEq eqType (taus, tau's)
```

When a data definition is typechecked or evaluated, the treatment of each value constructor depends on whether its type is a function type.

S477c. ⟨*recognition of function types* S477c⟩≡ (S496c)

```
fun isfuntype (FUNTY _) = true
  | isfuntype _         = false
```

T.2.3 Declarations and module types

A module type is either an export list, a module arrow, or allof a collection of module types (an intersection type). An export list may export four different forms of component.

S477d. ⟨*definition of* modty *for Molecule* S477d⟩≡ (S478c) S477e ▷

```
datatype modty
  = MTEXPORTS of (name * component) list
  | MTARROW   of (modident * modty) list * modty
  | MTALLOF   of modty list
and component
  = COMPVAL    of ty
  | COMPMANTY  of ty
  | COMPABSTY  of path
  | COMPMOD    of modty
```

Many, many operations in Chapter 9 operate on “rooted” entities. A suitable type is defined here, along with a function that extracts the root, and a functorial map.

S477e. ⟨*definition of* modty *for Molecule* S477d⟩+≡ (S478c) ◁ S477d S478a ▷

```
type 'a rooted = 'a * path
fun root (_, path) = path
fun rootedMap f (a, path) = (f a, path)
```

Names that can be bound in an environment include values, types, modules, module types, and overloaded names (a collection of values of different types).

S478a. ⟨*definition of* `modty` *for Molecule* S477d⟩+≡ (S478c) ◁S477e S478b▷

```
type print_string = string
datatype binding
  = ENVVAL    of ty
  | ENVMANTY  of ty
  | ENVMOD    of modty rooted
  | ENVOVLN   of ty list  (* overloaded name *)
  | ENVMODTY  of modty
```

An `ENVMOD` has a module identifier only if it is a *top-level* module and has been *elaborated*.

Bindings are closely related to declarations, but they are not quite the same thing.

S478b. ⟨*definition of* `modty` *for Molecule* S477d⟩+≡ (S478c) ◁S478a

```
datatype decl
  = DECVAL    of tyex
  | DECABSTY
  | DECMANTY  of tyex
  | DECMOD    of modtyx
  | DECMODTY  of modtyx  (* only at top level *)
and modtyx
  = MTNAMEDX   of name
  | MTEXPORTSX of (name * decl) located list
  | MTALLOFX   of modtyx located list
  | MTARROWX   of (name located * modtyx located) list * modtyx located
```

T.3 Abstract syntax and values

Abstract syntax and values are defined by chunks organized as follows:

S478c. ⟨*abstract syntax and values for Molecule* S478c⟩≡ (S475)

```
⟨paths for Molecule S476a⟩
⟨definition of ty for Molecule S477a⟩
⟨definition of modty for Molecule S477d⟩
type vcon = name path'
datatype pat = WILDCARD
             | PVAR     of name
             | CONPAT   of vcon * pat list
⟨definitions of exp and value for Molecule S479a⟩
val unitVal = CONVAL (PNAME "unit", [])
⟨definition of def for Molecule S479c⟩
⟨definition of unit_test for explicitly typed languages (from chunk 697b)⟩
  | CHECK_MTYPE of pathex * modtyx
⟨definition of xdef (shared) S214b⟩
val BugInTypeInference = BugInTypeChecking (* to make \uml utils work *)
⟨string conversion of Molecule values S525a⟩
⟨definition of patString for μML and μHaskell (from chunk 697b)⟩
⟨string conversion of Molecule types and module types S525d⟩
⟨prettyprinting of Molecule types and module types S527a⟩
⟨string conversion of Molecule's abstract syntax S529⟩
⟨utility functions on μML values (from chunk 697b)⟩
```

Expressions.

S479a. ⟨*definitions of* exp *and* value *for Molecule* S479a⟩≡ (S478c) S479b ▷

type exp

```
type overloading = int ref
type formal = name * tyex
datatype exp
  = LITERAL    of value
  | VAR        of pathex
  | VCONX      of vcon
  | CASE       of exp * (pat * exp) list   (* XXX pat needs to hold a path *)
  | IFX        of exp * exp * exp (* could be syntactic sugar for CASE *)
  | SET        of name * exp
  | WHILEX     of exp * exp
  | BEGIN      of exp list
  | APPLY      of exp * exp list * overloading
  | LETX       of let_flavor * (name * exp) list * exp
  | LETRECX    of ((name * tyex) * exp) list * exp
  | LAMBDA     of formal list * exp
  | MODEXP     of (name * exp) list    (* from body of a generic module *)
  | ERRORX     of exp list
  | EXP_AT     of srcloc * exp
and let_flavor = LET | LETSTAR
```

Like μML, Molecule deals in constructed data, functions, and a few primitive types. It also has module values.

S479b. ⟨*definitions of* exp *and* value *for Molecule* S479a⟩+≡ (S478c) ◁ S479a

type value

```
and value
  = CONVAL of vcon * value ref list
  | SYM  of name
  | NUM  of int
  | MODVAL of value ref env
  | CLOSURE   of lambda * value ref env
  | PRIMITIVE of primop
  | ARRAY     of value array
 withtype lambda = name list * exp
      and primop = value list -> value
```

The definition forms of Molecule are the definition forms of nano-ML, plus DATA, OVERLOAD, and three module-definition forms.

S479c. ⟨*definition of* def *for Molecule* S479c⟩≡ (S478c)

type def
type data_def

```
type modtyex = modtyx
datatype baredef
           = VAL    of name * exp
           | VALREC   of name * tyex * exp
           | EXP      of exp                                (* not in a module
           | QNAME    of pathex                             (* not in a module
           | DEFINE   of name * tyex * (formal list * exp)
           | TYPE     of name * tyex
           | DATA     of data_def
           | OVERLOAD of pathex list
           | MODULE   of name * moddef
           | GMODULE  of name * (name * modtyex) list * moddef
           | MODULETYPE of name * modtyex                   (* not in a module *)
and moddef = MPATH       of pathex
           | MPATHSEALED of modtyex * pathex
           | MSEALED     of modtyex * def list
           | MUNSEALED   of def list
  withtype data_def = name * (name * tyex) list
       and def = baredef located
```

BugInTypeChecking	S213c
type env	304
type modty	S477d
type name	303
type path'	S476b
type pathex	S476b
PNAME	S476b
type rooted	S477e
type ty	S477a
type tyex	S477a

A named value, type, or module can also be a component.

S480a. ⟨*converting bound entities to components* S480a⟩≡ (S496c)

```
fun asComponent (x, ENVVAL tau)     = SOME (x, COMPVAL tau)
  | asComponent (x, ENVMANTY tau)   = SOME (x, COMPMANTY tau)
  | asComponent (m, ENVMOD (mt, _)) = SOME (m, COMPMOD mt)
  | asComponent (_, ENVOVLN _) = NONE
  | asComponent (_, ENVMODTY _) = raise InternalError "module type as component"
```

A data definition is processed in much the same was as in μML. The major difference is that Molecule uses a single environment for all type information.

S480b. ⟨*definitions of* basis *and* processDef *for Molecule* S480b⟩≡ (S518b) S481a ▷

processDataDef : data_def * basis * interactivity -> basis

```
type basis = binding env * value ref env
val emptyBasis : basis = (emptyEnv, emptyEnv)
fun processDataDef (dd, (Gamma, rho), interactivity) =
  let val bindings      = typeDataDef (dd, TOPLEVEL, Gamma)
      val Gamma'        = Gamma <+> bindings
      val comps         = List.mapPartial asComponent bindings
        (* could convert first component to abstract type here XXX *)
      val (rho', vcons) = evalDataDef (dd, rho)
      val _ = if echoes interactivity then
                ⟨print the new type and each of its value constructors for Molecule S480c⟩
              else
                ()
  in  (Gamma', rho')
  end
```

S480c. ⟨*print the new type and each of its value constructors for Molecule* S480c⟩≡ (S480b)

```
let fun ddString (_, COMPMANTY _) = "*"  (* kind of the type *)
      | ddString (_, COMPVAL tau) = typeString tau
      | ddString _ = raise InternalError "component of algebraic data type"
    val (kind, vcon_types) =
      case map ddString comps
       of s :: ss => (s, ss)
        | [] => raise InternalError "missing kind string"
    val (T, _) = dd
    val tau = (case find (T, Gamma')
                 of ENVMANTY tau => tau
                  | _ => raise Match)
              handle _ => raise InternalError "datatype is not a type"
in  ( println (typeString tau ^ " :: " ^ kind)
    ; ListPair.appEq (fn (K, tau) => println (K ^ " : " ^ tau))
                     (vcons, vcon_types)
    )
end
```

As promised in the text, an overload declaration is processed as if it were a sequence of single-path declarations. Each one is done by internal function next.

S481a. ⟨*definitions of* basis *and* processDef *for Molecule* S480b⟩+≡ (S518b) ◁S480b S481c▷

```
fun processOverloading (ps, (Gamma, rho), interactivity) =
  let fun next (p, (Gamma, rho)) =
        let val (tau, first) =
              case pathfind (p, Gamma)
                of ENVVAL tau =>
                     (tau, okOrTypeError (firstArgType (pathexString p, tau)))
                 | c => ⟨can't overload a c S481b⟩
            val x = plast p

            val currentTypes =
              (case find (x, Gamma)
                 of ENVOVLN vals => vals
                  | _ => []) handle NotFound _ => []
            val newTypes = tau :: currentTypes
            val Gamma' = bind (x, ENVOVLN newTypes, Gamma)

            val newVals = extendOverloadTable (x, evalpath (p, rho), rho)
            val rho' = bind (x, ref (ARRAY newVals), rho)

            val _ = if echoes interactivity then
                      app print ["overloaded ", x, " : ", typeString tau, "\n"]
                    else
                      ()
        in  (Gamma', rho')
        end
  in  foldl next (Gamma, rho) ps
  end
```

S481b. ⟨*can't overload a* c S481b⟩≡ (S481a S512c)

```
raise TypeError ("only functions can be overloaded, but " ^ whatdec c ^
                 " " ^ pathexString p ^ " is not a function")
```

A basis has just two environments. The binding environment holds compile-time information (mostly types) about all named entities. The other environment holds the locations of all the named values.

S481c. ⟨*definitions of* basis *and* processDef *for Molecule* S480b⟩+≡ (S518b) ◁S481a S481d▷

```
type basis = binding env * value ref env
```

And now, all of the definitions. First, the DATA and overload definitions, which are implemented above.

S481d. ⟨*definitions of* basis *and* processDef *for Molecule* S480b⟩+≡ (S518b) ◁S481c S482▷

```
processDef : def * basis * interactivity -> basis
```

```
fun processDef ((loc, DATA dd), (Gamma, rho), interactivity) =
      atLoc loc processDataDef (dd, (Gamma, rho), interactivity)
  | processDef ((loc, OVERLOAD ps), (Gamma, rho), interactivity) =
      atLoc loc processOverloading (ps, (Gamma, rho), interactivity)
```

When a bare name is entered as a definition, the interpreter prints a response based on what the name stands for. Because module types generally don't fit on one line, processDef uses the prettyprinter defined in Appendix J (page S276).

S482. ⟨*definitions of* basis *and* processDef *for Molecule* S480b⟩+≡ (S518b) ◁S481d S483▷

```
| processDef ((loc, QNAME px), (Gamma, rho), interactivity) =
    let val c = pathfind (px, Gamma)
        val x = pathexString px
        fun respond doc = println (layout ppwidth (indent(2, agrp doc)))
        fun typeResponse ty = if x = ty then ["abstract type ", x]
                              else ["type ", x, " = ", ty]
        fun typeResponseDoc ty =
            if x = typeString ty then doc "abstract type " ^^ doc x
            else doc "type " ^^ doc x ^^ doc " = " ^^ typeDoc ty

        infixr 0 $
        fun f $ x = f x

        fun response (ENVVAL _) =
              raise InternalError "ENVVAL reached response"
          | response (ENVMANTY tau) =
              typeResponseDoc tau
          | response (ENVMOD (mt as MTARROW _, _)) =
              doc "generic module " ^^ doc x ^^ doc " :" ^/ mtDoc mt
          | response (ENVMOD (mt, _)) =
              doc "module " ^^ doc x ^^ doc " :" ^/ mtDoc mt
          | response (ENVMODTY mt) =
              doc "module type " ^^ doc x ^^ doc " =" ^/ mtDoc mt
          | response (ENVOVLN []) =
              raise InternalError "empty overloaded name"
          | response (ENVOVLN (tau :: taus)) =
              let val first_prefix = doc "overloaded "
                  val rest_prefix  = doc "         "
                        (* rest_prefix drops 2 for indent in `respond` *)
                  fun binding pfx tau =
                    indent (9,
                            agrp (pfx ^^ doc x ^^ doc " :" ^/+ typeDoc tau))
              in  vgrp (brkSep (binding first_prefix tau ::
                                map (binding rest_prefix) taus))
              end

        val _ =
          if echoes interactivity then
            case c
              of ENVVAL _ =>
                   ignore (processDef ((loc, EXP (VAR px)), (Gamma, rho), interactivity))
               | _ =>
                   respond (response c)
          else
            ()
    in  (Gamma, rho)
    end
```

All other definition forms are handled by the same logic, which invokes `typdef` and then `evaldef`. Most of the logic is devoted to printing the values, which uses the overloaded Molecule `print` function if it's overloaded at the right type, and otherwise uses the interpreter function `valueString`.

S483. ⟨*definitions of* `basis` *and* `processDef` *for Molecule* S480b⟩+≡ (S518b) ◁S482

```
| processDef ((loc, d), (Gamma, rho), interactivity) =
    let val bindings   = atLoc loc typdef (d, TOPLEVEL, Gamma)
        val Gamma      = Gamma <+> bindings
        val printer    = defPrinter (d, Gamma) interactivity
        val (rho, vs)  = atLoc loc evaldef (d, rho)

        fun callPrintExp i v = (* to be passed to eval when needed *)
          APPLY (VAR (PNAME (loc, "print")), [LITERAL v], ref i)

        fun printfun x tau v =
          (* print value v of type tau, which is named x *)
          let val resolved =
               (case find ("print", Gamma)
                  of ENVOVLN taus => resolveOverloaded ("print", tau, taus)
                   | _ => ERROR "no printer for tau")
               handle NotFound _ => ERROR "'print' not found"
          in  case resolved
                of OK (_, i) => ignore (eval (callPrintExp i v, rho))
                 | ERROR _ =>
                     case d
                       of EXP _ => print (valueString v)
                        | _ => case tau
                                 of FUNTY _ => print x
                                  | _       => print (valueString v)
          end

        val _ = if echoes interactivity then
                  (printer printfun vs; print "\n")
                else
                  ()
    in  (Gamma, rho)
    end
```

T.5 Predefined modules, module types, and functions

Predefined things include not only modules but also module types and a handful of functions.

S484a. ⟨*predefined Molecule types, functions, and modules* S484a⟩≡ S486d ▷

```
⟨Molecule's predefined module types S485a⟩
(define bool and ([b : bool] [c : bool]) (if b c b))
(define bool or  ([b : bool] [c : bool]) (if b b c))
(define bool not ([b : bool])            (if b (= 1 0) (= 0 0)))
(define int mod ([m : int] [n : int]) (- m (* n (/ m n))))
```

T.5.1 *Characters*

The definitions of type `t` and values `space` and `right-curly` appear in Chapter 9. Everything else is here.

S484b. ⟨*definition of module* `Char` S484b⟩≡

```
(module [Char : (exports [abstype t]
                         [new : (int -> t)]
                         [left-curly : t]
                         [right-curly : t]
                         [left-round : t]
                         [right-round : t]
                         [left-square : t]
                         [right-square : t]
                         [newline : t]
                         [space : t]
                         [semicolon : t]
                         [quotemark : t]
                         [=  : (t t -> bool)]
                         [!= : (t t -> bool)]
                         [print : (t -> unit)]
                         [println : (t -> unit)])]

  ⟨definitions inside module Char 558a⟩
  (define int new ([n : int]) n)
  (val newline      10)
  (val semicolon    59)
  (val quotemark    39)
  (val left-round    40)
  (val right-round   41)
  (val left-curly   123)
  (val left-square   91)
  (val right-square  93)

  (val = Int.=)
  (val != Int.!=)

  (val print Int.printu)
  (define unit println ([c : t]) (print c) (print newline))
)
```

T.5.2 Predefined module types

In addition to the ARRAY module type defined in Chapter 9 (chunk 529b), Molecule defines several other module types.

S485a. ⟨*Molecule's predefined module types* S485a⟩≡ (S484a) S485b ▷

```
(module-type PRINTS
   (exports [abstype t]
            [print : (t -> unit)]
            [println : (t -> unit)]))
```

S485b. ⟨*Molecule's predefined module types* S485a⟩+≡ (S484a) ◁ S485a S485c ▷

```
(module-type BOOL
   (exports [abstype t]
            [#f : t]
            [#t : t]))
 ;;;; omitted: and, or, not, similar?, copy, print, println
```

S485c. ⟨*Molecule's predefined module types* S485a⟩+≡ (S484a) ◁ S485b S485d ▷

```
(module-type SYM
   (exports [abstype t]
            [=  : (t t -> Bool.t)]
            [!= : (t t -> Bool.t)]))
 ;;;; omitted: hash, similar?, copy, print, println
```

T.5.3 Total order and relational predicates

S485d. ⟨*Molecule's predefined module types* S485a⟩+≡ (S484a) ◁ S485c S485e ▷

```
(module-type ORDER
  (exports [abstype t]
           [LESS : t]
           [EQUAL : t]
           [GREATER : t]))

(module [Order : ORDER]
  (data t
    [LESS : t]
    [EQUAL : t]
    [GREATER : t]))
```

S485e. ⟨*Molecule's predefined module types* S485a⟩+≡ (S484a) ◁ S485d S485f ▷

```
(module-type RELATIONS
  (exports [abstype t]
           [<  : (t t -> Bool.t)]
           [<= : (t t -> Bool.t)]
           [>  : (t t -> Bool.t)]
           [>= : (t t -> Bool.t)]
           [=  : (t t -> Bool.t)]
           [!= : (t t -> Bool.t)]))
```

The generic module Relations is given a module with a compare function, and it implements relational predicates.

S485f. ⟨*Molecule's predefined module types* S485a⟩+≡ (S484a) ◁ S485e

```
(generic-module [Relations : ([M : (exports [abstype t]
                                            [compare : (t t -> Order.t)])]
                               --m-> (allof RELATIONS
                                            (exports [type t M.t])))]
  (type t M.t)
  ⟨definitions of the six relational predicates S486a⟩
)
```

S486a. ⟨*definitions of the six relational predicates* S486a⟩≡ (S485f) S486b ▷

```
(define bool < ([x : t] [y : t])
   (case (M.compare x y)
      [Order.LESS #t]
      [_    #f]))
(define bool > ([x : t] [y : t])
   (case (M.compare y x)
      [Order.LESS #t]
      [_    #f]))
```

S486b. ⟨*definitions of the six relational predicates* S486a⟩+≡ (S485f) ◁ S486a S486c ▷

```
(define bool <= ([x : t] [y : t])
   (case (M.compare x y)
      [Order.GREATER #f]
      [_       #t]))
(define bool >= ([x : t] [y : t])
   (case (M.compare y x)
      [Order.GREATER #f]
      [_       #t]))
```

S486c. ⟨*definitions of the six relational predicates* S486a⟩+≡ (S485f) ◁ S486b

```
(define bool = ([x : t] [y : t])
   (case (M.compare x y)
      [Order.EQUAL #t]
      [_     #f]))
(define bool != ([x : t] [y : t])
   (case (M.compare x y)
      [Order.EQUAL #f]
      [_     #t]))
```

T.5.4 Resizeable arrays

To make the `ArrayList` module easy to download and play with, it is defined in its own file, `arraylist.mcl`.

S486d. ⟨*predefined Molecule types, functions, and modules* S484a⟩+≡ ◁ S484a S491a ▷

⟨*arraylist.mcl* S486e⟩

S486e. ⟨*arraylist.mcl* S486e⟩≡ (S486d)

```
(generic-module
   [ArrayList : ([Elem : (exports [abstype t])] --m-> (allof ARRAYLIST
                                                               (exports [type elem Elem.t])

   (module A (@m Array Elem))
   (module U (@m UnsafeArray Elem))
   (record-module Rep t ([elems : A.t]
                         [low-index : int]
                         [population : int]
                         [low-stored : int]))
   (type t Rep.t)
   (type elem Elem.t)

   ⟨definitions of operations in ArrayList S486f⟩

)
```

S486f. ⟨*definitions of operations in* `ArrayList` S486f⟩≡ (S486e) S487a ▷

```
(define t from ([i : int])
  (Rep.make (U.new 3) i 0 0))

(define int size ([a : t]) (Rep.population a))
```

S487a. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁ S486f S487b ▷

```
(define bool in-bounds? ([a : t] [i : int])
  (if (>= i (Rep.low-index a))
      (< (- i (Rep.low-index a)) (Rep.population a))
      #f))
```

S487b. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁ S487a S487c ▷

```
(define int internal-index ([a : t] [i : int])
  (let* ([k (+ (Rep.low-stored a) (- i (Rep.low-index a)))]
         [_ (when (< k 0) (error 'internal-error: 'array-index))]
         [n (A.size (Rep.elems a))]
         [idx (if (< k n) k (- k n))])
    idx))

(define elem at ([a : t] [i : int])
  (if (in-bounds? a i)
      (A.at (Rep.elems a) (internal-index a i))
      (error 'array-index-out-of-bounds)))

(define unit at-put ([a : t] [i : int] [v : elem])
  (if (in-bounds? a i)
      (A.at-put (Rep.elems a) (internal-index a i) v)
      (error 'array-index-out-of-bounds)))
```

S487c. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁ S487b S487d ▷

```
(define int lo     ([a : t]) (Rep.low-index a))
(define int nexthi ([a : t]) (+ (Rep.low-index a) (Rep.population a)))
```

S487d. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁ S487c S487e ▷

```
(define unit maybe-grow ([a : t])
  (when (>= (size a) (A.size (Rep.elems a)))
    (let* ([n  (A.size (Rep.elems a))]
           [n' (if (Int.= n 0) 8 (Int.* 2 n))]
           [new-elems (U.new n')]
           [start (lo a)]
           [limit (nexthi a)]
           [i 0]
           [_ (while (< start limit)      ; copy the elements
                (A.at-put new-elems i (at a start))
                (set i (+ i 1))
                (set start (+ start 1)))])
      (Rep.set-elems! a new-elems)
      (Rep.set-low-stored! a 0))))
```

S487e. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁ S487d S487f ▷

```
(define unit addhi ([a : t] [v : elem])
  (maybe-grow a)
  (let ([i (nexthi a)])
     (Rep.set-population! a (+ (Rep.population a) 1))
     (at-put a i v)))
```

S487f. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁ S487e S488a ▷

```
(define unit addlo ([a : t] [v : elem])
  (maybe-grow a)
  (Rep.set-population! a (+ (Rep.population a) 1))
  (Rep.set-low-index!  a (- (Rep.low-index a)  1))
  (Rep.set-low-stored! a (- (Rep.low-stored a) 1))
  (when (< (Rep.low-stored a) 0)
     (Rep.set-low-stored! a (+ (Rep.low-stored a) (A.size (Rep.elems a)))))
  (at-put a (Rep.low-index a) v))
```

S488a. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁S487f S488b▷

```
(define elem remhi ([a : t])
  (if (<= (Rep.population a) 0)
      (error 'removal-from-empty-array)
      (let* ([v (at a (- (nexthi a) 1))]
             [_ (Rep.set-population! a (- (Rep.population a) 1))])
        v)))
```

S488b. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁S488a S488c▷

```
(define elem remlo ([a : t])
  (if (<= (Rep.population a) 0)
      (error 'removal-from-empty-array)
      (let* ([v (at a (lo a))]
             [_ (Rep.set-population! a (- (Rep.population a) 1))]
             [_ (Rep.set-low-index! a (+ (lo a) 1))]
             [_ (Rep.set-low-stored! a (+ (Rep.low-stored a) 1))]
             [_ (when (Int.= (Rep.low-stored a) (A.size (Rep.elems a)))
                    (Rep.set-low-stored! a 0))])
        v)))
```

S488c. ⟨*definitions of operations in* ArrayList S486f⟩+≡ (S486e) ◁S488b

```
(define unit setlo ([a : t] [i : int])
  (Rep.set-low-index! a i))
```

T.6 Primitive functions, primitive modules, and the initial basis

T.6.1 Primitive functions and module values

Primitives for a module are placed into a list. Each element has a name, an ML value (of type `primop`), and a type. The element can be converted into a pair to be inserted into a component environment, and also a type-checking environment.

S488d. ⟨*list of (typed) primitives for each primitive module* S488d⟩≡ (S475) S488e▷

```
type primitive
compval : primitive -> name * component
decval  : primitive -> name * binding
```

```
type primitive = name * primop * ty
fun compval (x, v, ty) = (x, COMPVAL ty)
fun decval  (x, v, ty) = (x, ENVVAL  ty)
```

If a primitive module exports an atomic type, like `int` or `sym`, it also exports operations that implement equivalence tests and printing on values of that type. These operations are always built by `eqPrintPrims`. Argument `project` converts a Molecule value to a primitive ML value that admits equality.

S488e. ⟨*list of (typed) primitives for each primitive module* S488d⟩+≡ (S475) ◁S488d S489a▷

```
eqPrintPrims : ty -> (value -> ''a) -> primitive list
```

```
fun eqPrintPrims tau project =
  let val comptype = FUNTY ([tau, tau], booltype)
      val printtype = FUNTY ([tau], unittype)
      val u = unitVal
      fun comparison f =
        binaryOp (embedBool o (fn (x, y) => f (project x, project y)))
  in  ("similar?",  comparison op =,  comptype) ::
      ("dissimilar?",  comparison op =,  comptype) ::
      ("=",  comparison op =,  comptype) ::
      ("!=", comparison op <>, comptype) ::
      ("print",   unaryOp (fn x => (print   (valueString x); u)), printtype) ::
      ("println", unaryOp (fn x => (println (valueString x); u)), printtype) ::
      []
  end
```

Modules `Sym` and `Bool` export *only* equivalence and printing operations.

S489a. ⟨*list of (typed) primitives for each primitive module* S488d⟩+≡ (S475) ◁S488e S489b▷

```
symPrims  : primitive list
boolPrims : primitive list
```

```
val symPrims =
  eqPrintPrims symtype
               (fn SYM s => s
                 | _ => raise BugInTypeChecking "comparing non-symbols")
val boolPrims =
  eqPrintPrims booltype
               (fn CONVAL (K, []) => K
                 | _ => raise BugInTypeChecking "comparing non-Booleans")
```

Module `Int` exports all the integer primitives. The primitives are built from ML functions, using the higher-order functions defined here.

S489b. ⟨*list of (typed) primitives for each primitive module* S488d⟩+≡ (S475) ◁S489a S489c▷

```
fun asInt (NUM n) = n
  | asInt v = raise BugInTypeChecking ("expected a number; got " ^ valueString v)

fun comparison f = binaryOp (embedBool o f)
fun intcompare f =
      comparison (fn (NUM n1, NUM n2) => f (n1, n2)
                   | _ => raise BugInTypeChecking "comparing non-numbers")
fun wordOp f =
  arithOp (fn (n, m) => Word.toInt (f (Word.fromInt n, Word.fromInt m)))
fun unaryIntOp f = unaryOp (NUM o f o asInt)
fun unaryWordOp f = unaryIntOp (Word.toInt o f o Word.fromInt)
```

The values and types of the integer primitives are as follows.

S489c. ⟨*list of (typed) primitives for each primitive module* S488d⟩+≡ (S475) ◁S489b S490▷

```
intPrims : primitive list
```

```
val arithtype = FUNTY ([inttype, inttype], inttype)
val comptype  = FUNTY ([inttype, inttype], booltype)

val intPrims =
  ("+", arithOp op +,   arithtype) ::
  ("-", arithOp op -,   arithtype) ::
  ("*", arithOp op *,   arithtype) ::
  ("/", arithOp op div, arithtype) ::

  ("land", wordOp Word.andb, arithtype) ::
  ("lor", wordOp Word.orb,   arithtype) ::
  (">>u", wordOp Word.>>,    arithtype) ::
  (">>s", wordOp Word.~>>,   arithtype) ::
  ("<<",  wordOp Word.<<,    arithtype) ::

  ("of-int", unaryOp id,          FUNTY ([inttype], inttype)) ::
  ("negated", unaryIntOp ~,       FUNTY ([inttype], inttype)) ::
  ("lnot", unaryWordOp Word.notb, FUNTY ([inttype], inttype)) ::

  ("<",  intcompare op <,  comptype) ::
  (">",  intcompare op >,  comptype) ::
  ("<=", intcompare op <=, comptype) ::
  (">=", intcompare op >=, comptype) ::
  ("printu"
  , unaryOp (fn n => (printUTF8 (asInt n); unitVal))
  , FUNTY ([inttype], unittype)
  ) ::
  eqPrintPrims inttype
               (fn NUM n => n
                 | _ => raise BugInTypeChecking "comparing non-numbers")
```

Note the `eqPrintPrims` at the end; module `Int` also exports the equivalence and printing functions.

Next are the two sets of array primitives: one for `ArrayCore` and another for `UnsafeArray`. The types of these modules are also defined here.

S490. ⟨*list of (typed) primitives for each primitive module* S488d⟩+≡ (S475) ◁S489c

```
arrayPrims   : primitive list
uarrayPrims  : primitive list
arraymodtype  : modty
uarraymodtype : modty
```

```
local
  val arraypath = PNAME arraymodident
  val arrayarg  = genmodident "Elem"
  val argpath   = PNAME arrayarg
  val resultpath = PAPPLY (arraypath, [argpath])
  val elemtype   = TYNAME (PDOT (argpath, "t"))
  val arraytype  = TYNAME (PDOT (resultpath, "t"))

  val uninitialized = CONVAL (PNAME "uninitialized", [])

  fun protect f x = f x
    handle Size      => raise RuntimeError "array too big"
         | Subscript => raise RuntimeError "array index out of bounds"


  fun asArray (ARRAY a) = a
    | asArray _         = raise BugInTypeChecking "non-array value as array"
  fun arrayLeft f (a, x) = f (asArray a, x)
in
  val arrayPrims =
    ("size", unaryOp (NUM o Array.length o asArray), FUNTY ([arraytype], inttype)) ::
    ("new", binaryOp (fn (NUM n, a) => ARRAY (protect Array.array (n, a))
                       | _ => raise BugInTypeChecking "array size not a number"),
            FUNTY ([inttype, elemtype], arraytype)) ::
    ("empty", fn _ => ARRAY (Array.fromList []), FUNTY ([], arraytype)) ::
    ("at", binaryOp (fn (ARRAY a, NUM i) => protect Array.sub (a, i)
                      | _ => raise BugInTypeChecking "Array.at array or index"),
            FUNTY ([arraytype, inttype], elemtype)) ::
    ("at-put", fn [ARRAY a, NUM i, x] => (protect Array.update (a, i, x); unitVal)
                | _ => raise BugInTypeChecking "number/types of args to Array.at-put",
            FUNTY ([arraytype, inttype, elemtype], unittype)) ::
    []

  val arraymodtype : modty =
    MTARROW ([(arrayarg, MTEXPORTS [("t", COMPABSTY (PDOT (argpath, "t")))]  : modty)],
             MTEXPORTS (("t", COMPABSTY (PDOT (resultpath, "t"))) ::
                        ("elem", COMPMANTY elemtype) ::
                        map compval arrayPrims) : modty)

  val uarrayPrims =
    ("new", unaryOp (fn (NUM n) => ARRAY (protect Array.array (n, uninitialized))
                       | _ => raise BugInTypeChecking "array size not a number"),
            FUNTY ([inttype], arraytype)) ::
    []

  val uarraymodtype : modty =
    MTARROW ([(arrayarg, MTEXPORTS [("t", COMPABSTY (PDOT (argpath, "t")))]  : modty)],
             MTEXPORTS (("t", COMPABSTY (PDOT (resultpath, "t"))) ::
                        map compval uarrayPrims) : modty)
end
```

The full `Array` module is built from `ArrayCore`.

S491a. ⟨*predefined Molecule types, functions, and modules* S484a⟩+≡ ◁S486d S491b▷

```
(generic-module
   [Array : ([M : (exports [abstype t])] --m->
                (allof ARRAY (exports (type elem M.t))))]
   (module A (@m ArrayCore M))
   (type t A.t)
   (type elem M.t)
   (val new A.new)
   (val empty A.empty)
   (val at A.at)
   (val size A.size)
   (val at-put A.at-put))
```

Module `Ref` is implemented using an array of size 1.

S491b. ⟨*predefined Molecule types, functions, and modules* S484a⟩+≡ ◁S491a

```
(generic-module
   [Ref : ([M : (exports [abstype t])] --m->
                  (exports [abstype t]
                           [new : (M.t -> t)]
                           [!   : (t -> M.t)]
                           [:=  : (t M.t -> unit)]))]
  (module A (@m ArrayCore M))
  (type t A.t)
  (define t    new ([x : M.t])  (A.new 1 x))
  (define M.t  !   ([cell : t]) (A.at cell 0))
  (define unit :=  ([cell : t] [x : M.t]) (A.at-put cell 0 x)))
```

T.6.2 *Types exported by the primitive modules*

When typechecking literal expressions, the interpreter needs access to predefined types like `int` and `sym` (which are synonyms for `Int.t` and `Sym.t` respectively). To keep things simple, all those types are defined here, and likewise the module identifiers for the primitive modules.

S491c. ⟨*primitive module identifiers and types used to type literal expressions* S491c⟩≡ (S496c) S491d▷

```
val arraymodname = "Array"

val intmodident    = genmodident "Int"
val symmodident    = genmodident "Sym"
val boolmodident   = genmodident "Bool"
val unitmodident   = genmodident "Unit"
val arraymodident  = genmodident arraymodname
val uarraymodident = genmodident "UnsafeArray"

val inttype  = TYNAME (PDOT (PNAME intmodident, "t"))
val symtype  = TYNAME (PDOT (PNAME symmodident, "t"))
val booltype = TYNAME (PDOT (PNAME boolmodident, "t"))
val unittype = TYNAME (PDOT (PNAME unitmodident, "t"))
```

An array type is made using a path that instantiates the `Array` module.

S491d. ⟨*primitive module identifiers and types used to type literal expressions* S491c⟩+≡ (S496c) ◁S491c

```
fun arraytype tau =
  case tau
    of TYNAME (PDOT (module, "t")) =>
         TYNAME (PDOT (PAPPLY (PNAME arraymodident, [module]), "t"))
     | _ => raise InternalError "unable to form internal array type"
```

T.6.3 Building the initial basis

Each of the four atomic modules is given an export-list type, which consists of an abstract type `t` plus all the module's primitives. Each primitive is considered as a component, using `compval`. Each of the two the array modules has a type defined above.

S492a. ⟨*primitive modules and definition of* `initialBasis` S492a⟩≡ (S475) S492b ▷

`module : modident -> primitive list -> binding`

```
local
  fun module id primvals =
    let val typeT = ("t", COMPABSTY (PDOT (PNAME id, "t")))
    in  ENVMOD (MTEXPORTS (typeT :: map compval primvals), PNAME id)
    end
  val unitinfo = ("unit", unitVal, TYNAME (PDOT (PNAME unitmodident, "t")))
in
  val intmod  = module intmodident intPrims
  val symmod  = module symmodident symPrims
  val boolmod = module boolmodident boolPrims
  val unitmod = module unitmodident [unitinfo]
  val arraymod  = ENVMOD (arraymodtype,  PNAME arraymodident)
  val uarraymod = ENVMOD (uarraymodtype, PNAME uarraymodident)
end
```

The initial value of each module contains all the named components, each of which goes into a `ref` cell.

S492b. ⟨*primitive modules and definition of* `initialBasis` S492a⟩+≡ (S475) ◁ S492a S492c ▷

`intmodenv : value ref env`

```
fun addValWith f ((x, v, ty), rho) = bind (x, f v, rho)
val intmodenv    = foldl (addValWith (ref o PRIMITIVE)) emptyEnv intPrims
val arraymodenv  = foldl (addValWith (ref o PRIMITIVE)) emptyEnv arrayPrims
val boolmodenv   = foldl (addValWith (ref o PRIMITIVE)) emptyEnv boolPrims
val unitmodenv = bind ("unit", ref (CONVAL (PNAME "unit", [])), emptyEnv)
val symmodenv  = foldl (addValWith (ref o PRIMITIVE)) emptyEnv symPrims
```

And finally the modules themselves. Each has a type (expressed as a `binding`) and a value.

S492c. ⟨*primitive modules and definition of* `initialBasis` S492a⟩+≡ (S475) ◁ S492b S493a ▷

`modules : (name * binding * value) list`

```
val modules =
  let fun primExp (x, f, _) = (x, LITERAL (PRIMITIVE f))
  in  [ ("Int",  intmod,  MODVAL intmodenv)
      , ("Bool", boolmod, MODVAL boolmodenv)
      , ("Unit", unitmod, MODVAL unitmodenv)
      , ("Sym",  symmod,  MODVAL symmodenv)
      , (arraymodname,  arraymod,
         CLOSURE ((["Elem"], MODEXP (map primExp arrayPrims)), emptyEnv))
      , ("UnsafeArray",  uarraymod,
         CLOSURE ((["Elem"], MODEXP (map primExp uarrayPrims)), emptyEnv))
      , ("ArrayCore",  arraymod,
         CLOSURE ((["Elem"], MODEXP (map primExp arrayPrims)), emptyEnv))

      , ("#t", ENVVAL booltype, CONVAL (PNAME "#t", []))
      , ("#f", ENVVAL booltype, CONVAL (PNAME "#f", []))
      ]
  end
```

The initial basis is defined in two steps. First, all the primitive modules are added to an empty basis. Then the predefined things are added.

S493a. ⟨*primitive modules and definition of* `initialBasis` S492a⟩+≡ (S475) ◁S492c

```
initialBasis : basis
```

```
val emptyBasis = (emptyEnv, emptyEnv)
val initialBasis =
  let fun addmod ((x, dbl, v), (Gamma, rho)) =
        (bind (x, dbl, Gamma), bind (x, ref v, rho))
  in  foldl addmod emptyBasis modules
  end

val initialBasis =
  let val predefinedTypes =
            ⟨predefined Molecule types, functions, and modules, as strings (from chunk S484a)⟩
      val xdefs = stringsxdefs ("predefined types etc", predefinedTypes)
  in  readEvalPrintWith predefinedFunctionError
                          (xdefs, initialBasis, noninteractive)
  end
```

T.7 Paths and environments

T.7.1 Looking up path expressions

One complexity of Molecule's type system is that we continually find ourselves converting between *bindings* (which appear in environments) and *components* (which appear in export lists). Module components are not rooted, but module bindings are rooted—so to convert a component to a binding requires a root.

S493b. ⟨*path-expression lookup* S493b⟩≡ (S496c) S493c ▷

```
asBinding : component * path -> binding
uproot : binding -> component
```

```
fun asBinding (COMPVAL tau, root) = ENVVAL tau
  | asBinding (COMPABSTY path, root) = ENVMANTY (TYNAME path)
  | asBinding (COMPMANTY tau, root) = ENVMANTY tau
  | asBinding (COMPMOD mt, root) = ENVMOD (mt, root)

fun uproot (ENVVAL tau) = COMPVAL tau
  | uproot (ENVMANTY tau) = COMPMANTY tau
  | uproot (ENVMOD (mt, _)) = COMPMOD mt
  | uproot d = raise InternalError (whatdec d ^ " as component")
```

The fundamental operation on environments is looking up a *path*, not just a name. Internal function `mtfind` looks up the meaning of a component in a module type—which can then be converted to a binding.

S493c. ⟨*path-expression lookup* S493b⟩+≡ (S496c) ◁S493b S494b ▷

```
pathfind : pathex * binding env -> binding
```

```
fun pathfind (PNAME x, Gamma) = find (snd x, Gamma)
  | pathfind (PDOT (path, x), Gamma) =
      let ⟨definition of mtfind S495d⟩
      in  case pathfind (path, Gamma)
            of ENVMOD (mt, root) =>
                 (asBinding (valOf (mtfind (x, mt)), root) handle Option =>
                   noComponent (path, x, mt))
             | dec => ⟨tried to select path.x but path is a dec S494a⟩
      end
  | pathfind (PAPPLY (fpx, actualpxs) : pathex, Gamma) =
     ⟨instantiation of module fpx to actualpxs S494c⟩
and noComponent (path, x, mt) =
  raise TypeError ("module " ^ pathexString path ^ " does not have a component " ^
                   pathexString (PDOT (path, x)) ^ "; its type is " ^ mtString mt)
```

S494a. ⟨*tried to select* path.x *but* path *is a* dec S494a⟩≡ (S493c)

```
raise TypeError ("Tried to select " ^ pathexString (PDOT (path, x)) ^ ", but " ^
                 pathexString path ^ " is " ^ whatdec dec ^ ", which does not " ^
                 " have components")
```

Function pathfind can't look up a value constructor, because a value constructor is a path that has only a name, not a location. But a value constructor can be converted to a full pathex by adding a meaningless location.

S494b. ⟨*path-expression lookup* S493b⟩+≡ (S496c) ◁S493c S495e▷

pathexOfVcon : vcon -> pathex

```
local
  val vconLoc = ("unlocated value constructor", ~99)
in
  fun pathexOfVcon (PNAME x) = PNAME (vconLoc, x)
    | pathexOfVcon (PDOT (path, x)) = PDOT (pathexOfVcon path, x)
    | pathexOfVcon (PAPPLY _) = raise InternalError "application vcon"
end
```

Now we can tackle the two big pieces of pathfind. First, the instantiation of a generic module when it is applied to actual parameters. The instantiation is specified by Leroy's Apply rule. The idea is summarized as follows:

$$f : \Pi A{:}T.B \qquad\qquad f \mathbin{@@} M : B[A \mapsto M]$$

This rule works even if B is itself an arrow type. Uncurrying, it means that when substituting for the first formal parameter, I substitute in all the remaining formal parameters. The type of the instantiation the result type of module fpx *after* substitution.

S494c. ⟨*instantiation of module* fpx *to* actualpxs S494c⟩≡ (S493c)

```
let fun rootedModtype px = case pathfind (px, Gamma)
                             of ENVMOD (mt, root) => (mt, root)
                              | dec => notModule (dec, px)
    and notModule (dcl, px) =
      raise TypeError ("looking for a module, but " ^ pathexString px ^
                       " is a " ^ whatdec dcl)
    val (fmod, actuals) = (rootedModtype fpx, map rootedModtype actualpxs)
    val (formals, result) = case fmod
                              of (MTARROW fr, _) => fr
                               | _ => ⟨instantiated exporting module fpx S495a⟩
    fun resty ( []
              , []
              , result) =
          result
      | resty ( (formalid, formalmt) :: formals
              , (actmt,    actroot)  :: actuals
              , result) =
          let val theta = formalid |--> actroot
              fun fsubst (ident, mt) = (ident, mtsubstRoot theta mt)
              val mtheta = msubsn (actmt, actroot)
              val subst = mtsubstManifest mtheta o mtsubstRoot theta
          in  case implements (actroot, actmt, mtsubstRoot theta formalmt)
                of OK () => resty (map fsubst formals, actuals, subst result)
                 | ERROR msg => ⟨can't pass actroot as formalid to fpx S495b⟩
          end
      | resty _ = ⟨wrong number of arguments to fpx S495c⟩
in  ENVMOD (resty (formals, actuals, result), PAPPLY (root fmod, map root actuals))
end
```

Error messages for instantiation are as follows:

S495a. ⟨*instantiated exporting module* `fpx` S495a⟩≡ (S494c)

```
raise TypeError ("module " ^ pathexString fpx ^ " is an exporting module, and only " ^
                 " a generic module can be instantiated")
```

S495b. ⟨*can't pass* `actroot` *as* `formalid` *to* `fpx` S495b⟩≡ (S494c)

```
raise TypeError ("module " ^ pathString actroot ^ " cannot be used as argument " ^
                 modidentString formalid ^ " to generic module " ^ pathexString fpx ^
                 ": " ^ msg)
```

S495c. ⟨*wrong number of arguments to* `fpx` S495c⟩≡ (S494c)

```
raise TypeError ("generic module " ^ pathexString fpx ^ " is expecting " ^
                 countString formals "parameter" ^ ", but got " ^
                 countString actuals "actual parameter")
```

The other piece of `pathfind` looks up a name in a module type to find a component. Because I have not worked out the theory of intersection types in its entirety, there are some tricky cases that I have not implemented,

S495d. ⟨*definition of* `mtfind` S495d⟩≡ (S493c)

```
mtfind : name * modty -> component option
```

```
fun mtfind (x, mt as MTEXPORTS comps) : component option =
      (SOME (find (x, comps)) handle NotFound _ => NONE)
  | mtfind (x, MTARROW _) =
      raise TypeError ("tried to select component " ^ x ^
                       " from generic module " ^ pathexString path)
  | mtfind (x, mt as MTALLOF mts) =
     (case List.mapPartial (fn mt => mtfind (x, mt)) mts
        of [comp] => SOME comp
         | [] => NONE
         | comps =>
             let val abstract = (fn COMPABSTY _ => true | _ => false)
                 val manifest = (fn COMPMANTY _ => true | _ => false)
                 fun tycomp c = abstract c orelse manifest c
             in  if not (List.all tycomp comps) then
                   if List.exists tycomp comps then
                     raise BugInTypeChecking
                             "mixed type and non-type components"
                   else
                     unimp "value or module component in multiple signatures"
                 else
                   case List.filter manifest comps
                     of [comp] => SOME comp
                      | [] => SOME (hd comps)  (* all abstract *)
                      | _ :: _ :: _ =>
                          unimp ("manifest-type component " ^ x ^
                                 " in multiple signatures")
             end)
```

A common special case of `pathfind` is to look up the name of a module.

S495e. ⟨*path-expression lookup* S493b⟩+≡ (S496c) ◁S494b

```
findModule : pathex * binding env -> modty
```

```
fun findModule (px, Gamma) =
  case pathfind (px, Gamma)
    of ENVMOD (mt, _) => mt
     | dec => raise TypeError ("looking for a module, but " ^
                               pathexString px ^ " is a " ^ whatdec dec)
```

In type-checking code, it's frequently useful to refer to components of a module to things bound in the static environment. These functions are used for diagnostic messages throughout the type checker.

S496a. ⟨*environments for Molecule's defined names* S496a⟩≡ (S475) S496b ▷

```
fun whatcomp (COMPVAL _) = "a value"
  | whatcomp (COMPABSTY _) = "an abstract type"
  | whatcomp (COMPMANTY _) = "a manifest type"
  | whatcomp (COMPMOD _) = "a module"
```

S496b. ⟨*environments for Molecule's defined names* S496a⟩+≡ (S475) ◁ S496a S524b ▷

```
fun whatdec (ENVVAL _) = "a value"
  | whatdec (ENVMANTY _) = "a manifest type"
  | whatdec (ENVOVLN _) = "an overloaded name"
  | whatdec (ENVMOD _) = "a module"
  | whatdec (ENVMODTY _) = "a module type"
```

T.8 Implementation of Molecule's type system

Fasten your seat belt. The elements are as follows:

S496c. ⟨*type checking for Molecule* S496c⟩≡ (S475)

⟨*additional operations for composing* `error` *values* S498⟩
⟨`context` *for a Molecule definition* S509b⟩
⟨*type equality for Molecule* S477b⟩
⟨*recognition of function types* S477c⟩
⟨*substitutions for Molecule* S513c⟩
⟨*type components of module types* S497a⟩
⟨*utilities for module-type realization* S501c⟩
⟨*module-type realization* S500c⟩
⟨*invariants of Molecule* S496d⟩
⟨`implements` *relation, based on* `subtype` *of two module types* S499⟩
⟨*path-expression lookup* S493b⟩
⟨*converting bound entities to components* S480a⟩
⟨*elaboration of Molecule type syntax into types* S515c⟩
⟨*primitive module identifiers and types used to type literal expressions* S491c⟩
⟨*utility functions on Molecule types* S502b⟩
⟨`typeof` *a Molecule expression* (from chunk 697b)⟩
⟨*principal type of a module* S507b⟩
⟨*elaboration and evaluation of* `data` *definitions for Molecule* S511d⟩
⟨*elaborate a Molecule definition* S507c⟩

T.8.1 An invariant on combined module types

Important invariant of the least upper bound: In any *semantic* `MTALLOF`, if a type name appears as manifest in *any* alternative, it appears *only* as manifest, never as abstract—and the module type has no references to an abstract type of that name.

Violations of this invariant are detected by function `mixedManifestations`, which looks for abstract-type paths that include a manifest type.

S496d. ⟨*invariants of Molecule* S496d⟩≡ (S496c)

`mixedManifestations : modty -> bool`

```
fun mixedManifestations mt =
  let val path = PNAME (MODTYPLACEHOLDER "invariant checking")
      val manifests = msubsn (mt, path)
      val abstracts = abstractTypePaths (mt, path)
  in  List.exists (hasKey manifests) abstracts
  end
```

The abstract-type paths are enumerated here.

S497a. ⟨*type components of module types* S497a⟩≡ (S496c)

`abstractTypePaths : modty rooted -> path list`

```
  fun abstractTypePaths (MTEXPORTS cs, path : path) =
        let fun ats (t, COMPABSTY _) = [PDOT (path, t)]
              | ats (x, COMPMOD mt) = abstractTypePaths (mt, PDOT (path, x))
              | ats _ = []
        in  concatMap ats cs
        end
    | abstractTypePaths (MTALLOF mts, path) =
        concatMap (fn mt => abstractTypePaths (mt, path)) mts
    | abstractTypePaths (MTARROW _, _) = []   (* could be bogus, cf Leroy rule 21 *)
```

And the invariant is established by this smart constructor. It uses functions `msubsn` and `simpleSyntacticMeet` defined below.

S497b. ⟨*definition of smart constructor* `allofAt` S497b⟩≡ (S500c)

`allofAt : modty list rooted -> modty`

```
  fun allofAt (mts, path) =
    let val mt = MTALLOF mts
        val mantypes = msubsn (mt, path)
        val abstypes = abstractTypePaths (mt, path)
    in  if List.exists (hasKey mantypes) abstypes then
          simpleSyntacticMeet (mtsubstManifest mantypes mt)
        else
          mt
    end
```

In case an intersection type contains a mix of manifest and abstract definitions of the same type, the invariant can be restored by judicious use of `allofAt`. Function `unmixTypes` uses `allofAt` to restore the invariant.

S497c. ⟨*definition of* `unmixTypes` S497c⟩≡ (S500c)

`unmixTypes : modty rooted -> modty`

```
  fun unmixTypes (mt, path) =
    let fun mtype (MTEXPORTS cs) = MTEXPORTS (map comp cs)
          | mtype (MTALLOF mts)  = allofAt (map mtype mts, path)
          | mtype (MTARROW (args, result)) =
              MTARROW (map (fn (x, mt) => (x, mtype mt)) args, mtype result)
        and comp (x, COMPMOD mt) = (x, COMPMOD (unmixTypes (mt, PDOT (path, x))))
          | comp c = c
    in  mtype mt
    end
```

T.8.2 Module subtyping

The subtyping rules are reproduced in Figure T.1. To understand them, you must understand Leroy's substitutions.

The key goals of the implementation are as follows:

- If subtyping fails, a witness should be keyed by path.
- Error messages should tell the whole story, e.g., "context requires that `t` be both `int` and `bool`."
- If an intersection is uninhabited, ..."Try a cheap and cheerful solution to uninhabited intersections, e.g., incompatible manifest types?"

$$\boxed{\mathcal{T} <: \mathcal{T}'} \qquad \frac{\mathcal{T} <: \mathcal{T}_1' \qquad \mathcal{T} <: \mathcal{T}_2'}{\mathcal{T} <: \mathcal{T}_1' \wedge \mathcal{T}_2'} \qquad \frac{\texttt{comps}(\mathcal{T}) <: Ds'}{\mathcal{T} <: \textsc{exports}(Ds')}$$

$$\boxed{Ds <: Ds'} \qquad \frac{}{Ds <: [\,]}$$

$$\frac{Ds = Ds_{pre}, D, Ds_{post} \qquad D <: D'' \qquad Ds <: Ds''}{Ds <: (D'', Ds'')}$$

$$\boxed{D <: D'} \qquad \frac{}{t :: \lfloor * \rfloor_{\pi} <: t :: \lfloor * \rfloor_{\pi'}} \qquad \frac{}{t = \tau <: t :: \lfloor * \rfloor_{\pi'}} \qquad \frac{}{t = \tau <: t = \tau}$$

$$\frac{}{t :: \lfloor * \rfloor_{\pi} <: t = \pi} \qquad \frac{}{x : \tau <: x : \tau} \qquad \frac{\mathcal{T} <: \mathcal{T}'}{M : \mathcal{T} <: M : \mathcal{T}'}$$

Figure T.1: Subtyping

The result of a subtype test is represented not by a Boolean but by a value of type `unit error`. When more than error is detected, I use only the first one, by applying one of the following two functions:

S498. ⟨*additional operations for composing* `error` *values* S498⟩≡ (S496c)

```
>> : unit error * unit error -> unit error
firstE : unit error list -> unit error
```

```
infix 1 >>
fun (OK ()) >> c = c
  | (ERROR msg) >> _ = ERROR msg

fun firstE []      = OK ()
  | firstE (e::es) = e >> firstE es
```

Function `subtype` implements relation $\mathcal{T} <: \mathcal{T}'$, and `csubtype` implements relation $D <: D'$.

S499. ⟨`implements` *relation, based on* `subtype` *of two module types* S499⟩≡ (S496c) S500b ▷

```
csubtype : component * component -> unit error
subtype  : modty * modty -> unit error
```

```
fun subtype mts =
  let fun st (MTARROW (args, res), MTARROW (args', res')) =
            let fun contra ([], [], res') = st (res, res')
                  | contra ((x, tau) :: args, (x', tau') :: args', res') =
                       (* substitute x for x' *)
                       let val theta = mtsubstRoot (x' |--> PNAME x)
                       in  st (theta tau', tau) >>
                           contra (args, map (prightmap theta) args', theta res')
                       end
                  | contra _ =
                      ERROR "generic modules have different numbers of arguments"
            in  contra (args, args', res')
            end
        | st (MTARROW (args, _), _) =
            ERROR ("expected an exporting module but got one that takes " ^
                   countString args "parameter")
        | st (_, MTARROW (args, _)) =
            ERROR ("expected a module that takes " ^
                   countString args "parameter" ^ ", but got an exporting module")
        | st (mt, MTALLOF mts') =
            firstE (map (fn mt' => st (mt, mt')) mts')
        | st (mt, MTEXPORTS comps') =
            compsSubtype (components mt, comps')
      and components (MTEXPORTS cs) = cs
        | components (MTALLOF mts) = concatMap components mts
        | components (MTARROW _) = raise InternalError "meet of arrow types"
      and compsSubtype (comps, comps') =
            let fun supplied (x, _) = List.exists (fn (y, _) => x = y) comps
                val (present, absent) = List.partition supplied comps'
                fun check (x, supercomp) =
                  let ⟨definition of csubtype S500a⟩
                  in  csubtype (find (x, comps), supercomp)
                  end
                    handle NotFound y =>
                      raise InternalError "missed present component"
                fun missingComponent (x, c) = x ^ " (" ^ whatcomp c ^ ")"
                val missedMsg =
                  if null absent then OK ()
                  else
                    ERROR ("an interface expected some components that are " ^
                           "missing: " ^
                           commaSep (map missingComponent absent))
            in  firstE (map check present) >> missedMsg
            end
  in  st mts
  end
```

S500a. ⟨*definition of* `csubtype` S500a⟩≡ (S499)

```
                                   csubtype : component * component -> unit error
```

```
fun csubtype (COMPVAL tau, COMPVAL tau') =
      if eqType (tau, tau') then OK ()
      else ERROR ("interface calls for value " ^ x ^ " to have type " ^
                  typeString tau' ^ ",\n           but it has type " ^ typeString tau)
  | csubtype (COMPABSTY _, COMPABSTY _) = OK ()  (* OK without comparing paths? *)
  | csubtype (COMPMANTY _, COMPABSTY _) = OK ()  (* likewise? *)
  | csubtype (COMPMANTY tau, COMPMANTY tau') =
      if eqType (tau, tau') then OK ()
      else ERROR ("interface calls for type " ^ x ^ " to manifestly equal " ^
                  typeString tau' ^ ",\n           but it is " ^ typeString tau)
  | csubtype (COMPABSTY path, COMPMANTY tau') =
      if eqType (TYNAME path, tau') then OK ()
      else ERROR ("interface calls for type " ^ x ^ " to manifestly equal " ^
                  typeString tau' ^ ",\n           but it is " ^ typeString (TYNAME path))
  | csubtype (COMPMOD m, COMPMOD m') =
      subtype (m, m')
  | csubtype (c, c') =
      ERROR ("interface calls for " ^ x ^ " to be " ^ whatcomp c' ^
             ",\n           but implementation provides " ^ whatcomp c)
```

Function `implements` tells if a subtype implements a supertype. What's the path for? It is the first argument to functions `msubsn` and to `abstractTypePaths`. Which means it's used as the prefix to produce the correct substitution, and that's it.

If you've read Leroy (2000), function `implements` is my approximation of Leroy's `modtype_match`. Instead of placing type equalities in an environment, I substitute.

S500b. ⟨`implements` *relation, based on* `subtype` *of two module types* S499⟩+≡ (S496c) ◁ S499

```
                                   implements : path * modty * modty -> unit error
```

```
fun implements (p, submt, supermt) =
  let val theta = msubsn (submt, p)
  in  subtype (submt, mtsubstManifest theta supermt)
  end
```

T.8.3 Realization of module types

The essences of module-type realization is substitution of manifest types for abstract types, as implemented by function `msubsn`. Substitution is also used to implement intersection (module) types, since the intersection of a manifest type with an abstract type requires substituting for the abstract type. All functions related to this substitution are therefore grouped here.

S500c. ⟨*module-type realization* S500c⟩≡ (S496c)

⟨*definition of* `msubsn` S501a⟩
⟨*definition of* `simpleSyntacticMeet` S501b⟩
⟨*definition of smart constructor* `allofAt` S497b⟩
⟨*definition of* `unmixTypes` S497c⟩

Realization substitutes manifest types for abstract types in a module type. Function msubsn is specified in Figure 9.16 in Chapter 9, page 567.

S501a. ⟨*definition of* msubsn S501a⟩≡ (S500c)

`msubsn : modty rooted -> tysubst`

```
fun msubsn (MTEXPORTS cs, path) =
      let fun mts (x, COMPMANTY tau) = [(PDOT (path, x), tau)]
            | mts (x, COMPMOD mt) = msubsn (mt, PDOT(path, x))
            | mts _ = []
      in  concatMap mts cs
      end
  | msubsn (MTALLOF mts, path) =
      concatMap (fn mt => msubsn (mt, path)) mts
  | msubsn (MTARROW _, path) = []   (* could be bogus, cf Leroy rule 21 *)
```

Quite often an allof type can be rewritten as an exports type. The game is simply to combine the export lists of the intersected module types. This combination is accomplished by filtering out redundant manifest-type declarations, then dropping any module type that consists only of redundant declarations (or is otherwise empty). This heuristic makes a type look nicer without changing its meaning.

S501b. ⟨*definition of* simpleSyntacticMeet S501b⟩≡ (S500c)

`simpleSyntacticMeet : modty -> modty`

```
val simpleSyntacticMeet =
  let val path = PNAME (MODTYPLACEHOLDER "syntactic meet")
      fun filterManifest (prev', []) = rev prev'
        | filterManifest (prev', mt :: mts) =
            let val manifests = msubsn (MTALLOF prev', path)
                fun redundant (COMPMANTY tau, p) =
                      (case associatedWith (p, manifests)
                         of SOME tau' => eqType (tau, tau')
                          | NONE => false)
                  | redundant _ = false
            in  filterManifest (filterdec (not o redundant) (mt, path) :: prev',
            end
      val filterManifest = fn mts => filterManifest ([], mts)
      fun mtall [mt] = mt
        | mtall mts  = MTALLOF mts
      val meet = mtall o List.filter (not o emptyExports) o filterManifest
  in  fn (MTALLOF mts) => meet mts
       | mt => mt
  end
```

Function filterdec selects only those components accepted by a predicate.

S501c. ⟨*utilities for module-type realization* S501c⟩≡ (S496c) S502a ▷

`filterdec : (component * path -> bool) -> modty rooted -> modty`

```
fun filterdec p (MTARROW f, path) = MTARROW f
  | filterdec p (MTALLOF mts, path) =
      MTALLOF (map (fn mt => filterdec p (mt, path)) mts)
  | filterdec p (MTEXPORTS xcs, path) =
      let fun cons ((x, c), xcs) =
            let val path = PDOT (path, x)
                val c = case c
                          of COMPMOD mt => COMPMOD (filterdec p (mt, path))
                           | _ => c
            in  if p (c, path) then
                  (x, c) :: xcs
                else
                  xcs
            end
      in  MTEXPORTS (foldr cons [] xcs)
      end
```

The module type (exports) is an identity of $\wedge$, so it is filtered out of `MTALLOF`.

S502a. ⟨*utilities for module-type realization* S501c⟩+≡ (S496c) ◁S501c

```
emptyExports : modty -> bool
```

```
fun emptyExports (MTEXPORTS []) = true
  | emptyExports _ = false
```

T.8.4 Type checking for expressions, with overloading

Type checking for expressions is much like type checking in Typed Impcore or Typed μScheme, except that Molecule supports operator overloading. It works like this:

- Only functions can be overloaded, and the overloading is resolved by consulting the type of the first argument.
- An overloaded name is associated with a *sequence* of values: one for each type at which the name is overloaded.
- At run time, the sequence is represented by an array of values.
- At compile time, the sequence is represented by a list of types.
- Adding an overloading means consing on to the front of the sequence.
- Using an overloaded name requires an index into the sequence. The first matching type wins.
- An overloaded name can be *used* only in a function application. At every application, therefore, the type checker writes the sequence index into the AST node.

An attempt to overload a name starts with a check for the type of the first argument.

S502b. ⟨*utility functions on Molecule types* S502b⟩≡ (S496c) S502c ▷

```
fun firstArgType (x, FUNTY (tau :: _, _)) = OK tau
  | firstArgType (x, FUNTY ([], _)) =
      ERROR ("function " ^ x ^ " cannot be overloaded because " ^
             "it does not take any arguments")
  | firstArgType (x, _) =
      ERROR (x ^ " cannot be overloaded because it is not a function")
```

Function `okOrTypeError` converts an `ERROR` value to a `TypeError` exception. And `ok` asserts that there should be no `ERROR` value.

S502c. ⟨*utility functions on Molecule types* S502b⟩+≡ (S496c) ◁S502b S503a ▷

```
okOrTypeError : 'a error -> 'a
ok            : 'a error -> 'a
```

```
fun okOrTypeError (OK a) = a
  | okOrTypeError (ERROR msg) = raise TypeError msg

fun ok a = okOrTypeError a
           handle _ => raise InternalError "overloaded non-function?"
```

To resolve overloaded name `f`, internal function `findAt` is given a list of the types at which name `f` is overloaded. It returns the type and index of the overloading in which the first argument is `argty`. (Because only functions of at least one argument can be overloaded, the argument type is extracted from `firstArgType` using `ok`.)

S503a. ⟨*utility functions on Molecule types* S502b⟩+≡ (S496c) ◁S502c

```
                    resolveOverloaded : name * ty * ty list -> (ty * int) error
fun resolveOverloaded (f, argty, tys) =
  let fun findAt (tau :: taus, i) =
            if eqType (argty, ok (firstArgType (f, tau))) then
              OK (tau, i)
            else
              findAt (taus, i + 1)
        | findAt ([], _) =
            ERROR ("cannot figure out how to resolve overloaded name " ^
                   f ^ " when applied to first argument of type " ^
                   typeString argty ^ " (resolvable: " ^
                   commaSep (map typeString tys) ^ ")")
  in  findAt (tys, 0)
  end
```

The `typeof` function is very much like the one for Typed μScheme, which is a homework exercise. To avoid spoiling that exercise, this appendix includes only the code for typechecking function applications.

S503b. ⟨`typeof` *a Molecule expression* **⟦prototype⟧** S503b⟩≡

```
                                             typeof : exp * binding env -> ty
                                             ty     : exp               -> ty
fun typeof (e, Gamma) =
  let fun ty e = typeof (e, Gamma)  (* replace with your code *)
      ⟨definitions of internal functions for typeof S503c⟩
  in  raise LeftAsExercise "typeof"  (* call ty e *)
  end
```

Function `typeofFunction` resolves any overloading. It takes the expression being applied, plus the types of the actual parameters. It returns the type of the expression, plus an index into the overloading table for this `APPLY` node. If the function is not overloaded, the index that comes back is `notOverloadedIndex`.

S503c. ⟨*definitions of internal functions for* `typeof` S503c⟩≡ (S503b) S504a▷

```
                                           typeofFunction : exp * ty list -> ty * int
fun typeofFunction (f, []) = (typeof (f, Gamma), notOverloadedIndex)
  | typeofFunction (f, first::_) =
      let fun resolve (EXP_AT (loc, f)) = atLoc loc resolve f
            | resolve (e as VAR (PNAME (_, f))) =
                (case find (f, Gamma)
                   of ENVOVLN taus =>
                           okOrTypeError (resolveOverloaded (f, first, taus))
                    | _ => (typeof (e, Gamma), notOverloadedIndex))
            | resolve e = (typeof (e, Gamma), notOverloadedIndex)
      in  resolve f
      end
```

The code for APPLY checks the argument types as expected. The only oddity is that it also has the side effect of updating the index field in the abstract-syntax tree. When an overloaded function is applied, that index is used to find its value at this call site.

Function maybeNamed adds f's name to a string, provided f has a name. And if things go wrong, diagnoseArgs compares the types of the formals and the actuals, producing a type-error message.

S504a. ⟨*definitions of internal functions for* typeof S503c⟩+≡ (S503b) ◁S503c

```
maybeNamed : string -> string
diagnoseArgs : ty list * ty list -> string
```

```
fun typeOfApply f actuals index =
  let val atys = map ty actuals
      ⟨definitions of maybeNamed and diagnoseArgs S504b⟩
  in  case typeofFunction (f, atys)
        of (FUNTY (formals, result), i) =>
             if eqTypes (atys, formals) then
                 result before index := i
             else
                 raise TypeError (diagnoseArgs (formals, atys))
         | _ => raise TypeError ("tried to apply " ^ maybeNamed "non-function" ^
                                 " of type " ^ typeString (ty f))
  end
```

Error checking is hell.

S504b. ⟨*definitions of* maybeNamed *and* diagnoseArgs S504b⟩≡ (S504a)

```
val got = (* what we actually got in the way of argument types *)
  case atys
    of [tau] => "argument of type " ^ typeString tau
     | []    => "no arguments"
     | _     => "arguments of types " ^ spaceSep (map typeString atys)
val actuals_are =
  case actuals of [_] => "argument"
                | _ => "arguments"

fun ordinal 1 = "first"
  | ordinal 2 = "second"
  | ordinal 3 = "third"
  | ordinal n = intString n ^ "th"

fun maybeNamed whattype =
  case stripExpAt f of VAR _ => whattype ^ " " ^ expString f
                     | _ => whattype

fun diagnoseArgs (formals, actuals) =
  let fun go (_, [], []) = raise InternalError "can't find arg fault"
        | go (k, f::fs, a::a's) =
            if eqType (f, a) then
              go (k + 1, fs, a's)
            else
              maybeNamed "function" ^ " expects " ^ ordinal k ^
              " argument of type " ^ typeString f ^ ", but got " ^ typeString a
        | go _ =
              maybeNamed "function" ^ " expects " ^
              countString formals "argument" ^ " but got " ^
              intString (length actuals)
  in  go (1, formals, actuals)
  end
```

$$\boxed{\Gamma \vdash e : \tau} \qquad \overset{\textsc{Case}}{\frac{\begin{array}{c}\Gamma \vdash e : \tau \\ \Gamma \vdash \texttt{[}p_i\ \ e_i\texttt{]} : \tau \rightarrow \tau_i, \quad 1 \le i \le n \\ \tau_1 = \cdots = \tau_n\end{array}}{\Gamma \vdash \textsc{case}(e, \texttt{[}p_1\ \ e_1\texttt{]}, \ldots, \texttt{[}p_n\ \ e_n\texttt{]}) : \tau_1}} \qquad \overset{\textsc{Vcon}}{\frac{\Gamma(K) = \tau}{\Gamma \vdash K : \tau}}$$

$$\boxed{\Gamma \vdash \texttt{[}p\ \ e\texttt{]} : \tau \rightarrow \tau'} \qquad \overset{\textsc{Choice}}{\frac{\Gamma, \Gamma' \vdash p : \tau \qquad \Gamma + \Gamma' \vdash e : \tau'}{\Gamma \vdash \texttt{[}p\ \ e\texttt{]} : \tau \rightarrow \tau'}}$$

$$\boxed{\Gamma, \Gamma' \vdash p : \tau} \qquad \overset{\textsc{PatVcon}}{\frac{\begin{array}{c}\Gamma \vdash K : \tau_1 \times \cdots \times \tau_k \rightarrow \tau \\ \Gamma, \Gamma'_i \vdash p_i : \tau_i, \quad 1 \le i \le k \\ \Gamma' = \Gamma'_1 \uplus \cdots \uplus \Gamma'_k\end{array}}{\Gamma, \Gamma' \vdash (K\ p_1\ \ \cdots\ \ p_k) : \tau}} \qquad \overset{\textsc{PatBareVcon}}{\frac{\Gamma \vdash K : \tau}{\Gamma, \{\} \vdash K : \tau}}$$

$$\overset{\textsc{PatWildcard}}{\frac{}{\Gamma, \{\} \vdash \textsc{wildcard} : \tau}} \qquad \overset{\textsc{PatVar}}{\frac{}{\Gamma, \{x \mapsto \tau\} \vdash x : \tau}}$$

Figure T.2: Typing rules for monomorphic case expressions, choices, and patterns

The type checking of a `CASE` expression is similar to what you would see in Chapter 8, but since it's not part of any homework exercise, I don't mind showing the version for Molecule. The right-hand side of each choice must have the same type, which is the type of the whole `CASE` expression. The type of a right-hand side is determined by the type of the scrutinee and the environment introduced by the pattern on the left. And there must be at least one choice.

S505. ⟨*type of* `CASE (e, choices)` S505⟩≡

```
let val tau = typeof (e, Gamma)  (* type of scrutinee *)
    ⟨definition of function patenv for Molecule S506b⟩

    fun choiceRtype (p, e) = (* type of the RHS of a choice *)
      let val Gamma' = patenv (p, Gamma, tau)
      in  typeof (e, Gamma <+> Gamma')
      end

    val rights = map choiceRtype choices
    ⟨definition of commonRightsType S506a⟩

in  commonRightsType rights
end
```

`countString`	S214d
`eqType`	S477b
`eqTypes`	S477b
`expString`	S529
`FUNTY`	S477a
`InternalError`	S219e
`intString`	S214c
`spaceSep`	S214e
`stripExpAt`	S529
`ty`	S503b
`TypeError`	S213c
`typeofFunction`	S503c
`typeString`	S526b
`VAR`	S479a

Function commonRightsType compares all the types for equality, and in case one doesn't match, it tracks the choice number n for use in an error message.

S506a. ⟨*definition of* commonRightsType S506a⟩≡ (S505)

```
fun badChoice n msg =
      raise TypeError ("in choice " ^ intString n ^
                       " of case expression, " ^ msg)

fun commonRightsType [] =
      raise TypeError "empty case expression cannot be assigned a type"
  | commonRightsType (firstright :: rights) =
      let fun check ([], _) = firstright
            | check (r::rs, n) =
                if eqType (r, firstright) then
                  check (rs, n + 1)
                else
                  badChoice n ("right-hand side has type " ^ typeString r ^
                               ", which does not match first right-hand side " ^
                               "(of type " ^ typeString firstright ^ ")")
      in  check (rights, 2)
      end
```

The type of a value constructor is found in the environment.

S506b. ⟨*definition of function* patenv *for Molecule* S506b⟩≡ (S505) S506c ▷

```
fun pvconType (K, Gamma) =
  (case pathfind (pathexOfVcon K, Gamma)
     of ENVVAL tau => tau
      | comp => raise TypeError (vconString K ^ " is not a value constructor"))
     handle NotFound x => raise TypeError ("no value constructor named " ^ x)
```

Function patenv introduces one binding into the environment for each variable in the pattern. It also checks applications of value constructors.

S506c. ⟨*definition of function* patenv *for Molecule* S506b⟩+≡ (S505) ◁ S506b

```
fun patenv (WILDCARD, _, tau) =
      emptyEnv
  | patenv (PVAR x, _, tau) =
      bind (x, ENVVAL tau, emptyEnv)
  | patenv (CONPAT (K, pats), Gamma, tau) =
      let fun patenvs ([], []) =
                []
            | patenvs (p::ps, tau::taus) =
                patenv(p, Gamma, tau) :: patenvs(ps, taus)
            | patenvs _ =
                raise TypeError ("wrong number of arguments " ^
                                 "to value constructor " ^ vconString K)
          ⟨definition of function badK S507a⟩
      in  case (pats, pvconType (K, Gamma))
            of ([], tau') => if eqType (tau, tau') then emptyEnv
                             else badK "type" tau'
             | (_, FUNTY (args, res)) =>
                 if eqType (tau, res) then
                   let val Gamma's = patenvs (pats, args)
                   in  disjointUnion Gamma's
                   end
                 else
                   badK "result type" res
             | (_, tau') =>
                 raise TypeError ("value constructor " ^ vconString K ^
                                  " is applied to patterns, but its type " ^
                                  typeString tau' ^ " is not a function type")
      end
```

$$\boxed{E \vdash \pi : \mathcal{T}} \qquad \frac{E \ni \pi : \lfloor \mathcal{T}' \rfloor_{\pi'}}{E \vdash \pi : \mathcal{T}'/\pi'}$$

$$\begin{aligned}
(\mathcal{T}_1 \land \mathcal{T}_2)/\pi &= \mathcal{T}_1/\pi \land \mathcal{T}_2/\pi & (t = \tau)/\pi &= (t = \tau)\\
\text{EXPORTS}(Ds)/\pi &= \text{EXPORTS}(Ds/\pi) & (t :: \lfloor * \rfloor_{\pi'})/\pi &= (t = \pi.t)\\
[\,]/\pi &= [\,] & (x : \tau)/\pi &= x : \tau\\
(D, Ds)/\pi &= D/\pi, Ds/\pi & (M : \mathcal{T})/\pi &= x : \mathcal{T}/\pi.M
\end{aligned}$$

Figure T.3: Strengthening module types and components

If a value constructor has the wrong type or the wrong result type, function `badK` issues this error message:

S507a. ⟨*definition of function* `badK` S507a⟩≡ (S506c)

```
fun badK what tau' =
  raise TypeError ("expected pattern with type " ^ typeString tau ^
                   ", but found value constructor " ^ vconString K ^
                   " with " ^ what ^ " " ^ typeString tau')
```

T.8.5 Typechecking modules: strengthening

Strengthening a module type converts every abstract type into a manifest type, which is manifestly equal to itself. For “itself” to mean anything, we have to know the path associated with the module type—that is, the module type has to be rooted.

S507b. ⟨*principal type of a module* S507b⟩≡ (S496c)

```
strengthen : modty rooted -> modty
```

```
fun strengthen (MTEXPORTS comps, p) =
      let fun comp (c as (x, dc)) =
            case dc
              of COMPABSTY _ => (x, COMPMANTY (TYNAME (PDOT (p, x))))
               | COMPMOD mt  => (x, COMPMOD (strengthen (mt, PDOT (p, x))))
               | COMPVAL   _ => c
               | COMPMANTY _ => c
      in  MTEXPORTS (map comp comps)
      end
  | strengthen (MTALLOF mts, p) =
      allofAt (map (fn mt => strengthen (mt, p)) mts, p)
  | strengthen (mt as MTARROW _, p) =
      mt
```

allofAt	S497b
COMPABSTY	S477d
COMPMANTY	S477d
COMPMOD	S477d
COMPVAL	S477d
type modty	S477d
MTALLOF	S477d
MTARROW	S477d
MTEXPORTS	S477d
type name	303
PDOT	S476b
type rooted	S477e
type ty	S477a
TYNAME	S477a

T.8.6 Responding to definitions

The interpreter issue a response to each top-level definition. A response has type `value_printer`, and it may be formed from a list of strings.

S507c. ⟨*elaborate a Molecule definition* S507c⟩≡ (S496c) S508a ▷

```
type value_printer
printStrings : string list -> value_printer
```

```
type value_printer =
  interactivity -> (name -> ty -> value -> unit) -> value list -> unit
fun printStrings ss _ _ vs =
  app print ss
```

A response may also be formed from a doc, which is a type that contains instructions for prettyprinting (automated indentation and such), as described on page S276 (Appendix J). Prettyprinting is done 77 columns wide, unless environment variable COLS says otherwise.

S508a. ⟨*elaborate a Molecule definition* S507c⟩+≡ (S496c) ◁S507c S508b▷

`printDoc : doc -> value_printer`

```
val ppwidth =
  getOpt(Option.mapPartial Int.fromString (OS.Process.getEnv "COLS"), 77)

fun printDoc doc interactivity _ _ =
  let val margin = if prompts interactivity then 2 else 0
  in  print (layout ppwidth (indent (margin, agrp doc)))
  end
```

The response to a definition includes the name of the thing defined, which is computed by defName.

S508b. ⟨*elaborate a Molecule definition* S507c⟩+≡ (S496c) ◁S508a S508c▷

`defName : baredef -> string`

```
fun defName (VAL (x, _)) = x
  | defName (VALREC (x, _, _)) = x
  | defName (EXP _) = "it"
  | defName (QNAME _) = raise InternalError "defName QNAME"
  | defName (DEFINE (x, _, _)) = x
  | defName (TYPE (t, _)) = t
  | defName (DATA (t, _)) = raise InternalError "defName DATA"
  | defName (OVERLOAD _) = raise InternalError "defName OVERLOAD"
  | defName (MODULE (m, _)) = m
  | defName (GMODULE (m, _, _)) = m
  | defName (MODULETYPE (t, _)) = t
```

Once name and binding are known, a response is computed by defReponse.

S508c. ⟨*elaborate a Molecule definition* S507c⟩+≡ (S496c) ◁S508b S509a▷

`defResponse : name * binding -> value_printer`

```
fun printMt what m how mt =
  printDoc (doc (concat [what, " ", m, " ", how]) ^/+ mtDoc mt)

fun defResponse (x, c) =
  case c
    of ENVVAL tau =>
         (fn _ => fn printfun =>
             fn [v] => (printfun x tau v; app print [" : ", typeString tau])
                       (* can't do better b/c printfun only prints *)
              | _ => raise InternalError "value count for val definition")
     | ENVMANTY tau =>
         let val expansion = typeString tau
         in  if x = expansion then
               printStrings ["abstract type ", x]
             else
               printDoc (doc "type" ^^ doc x ^^ doc "=" ^/+ typeDoc tau)
         end
     | ENVMOD (mt as MTARROW _, _) => printMt "generic module" x ":" mt
     | ENVMOD (mt, _)              => printMt "module" x ":" mt
     | ENVMODTY mt                 => printMt "module type" x "=" mt
     | ENVOVLN _ => raise InternalError "defResponse to overloaded name"
```

And defPrinter composes defName and defResponse.

S509a. ⟨*elaborate a Molecule definition* S507c⟩+≡ (S496c) ◁S508c S509c▷

```
defPrinter : baredef * binding env -> value_printer
```

```
fun defPrinter (d, Gamma) =
      let val x = defName d
      in  defResponse (x, find (x, Gamma))
          handle NotFound _ => raise InternalError "defName not found"
      end
```

T.8.7 Typechecking definitions

A definition can appear at top level or inside a module body. When a module definition appears at top level, its path is formed with a fresh module identifier. The identifier is generated by genmodident, which guarantees the uniqueness of the new top-level module. But when a module is defined inside another module, its path is computed by combining its name with the path of the module in which it is defined. No new unique identifier is necessary.

The place where a module definition (or any other form of definition) appears is represented by a context.

S509b. ⟨context *for a Molecule definition* S509b⟩≡ (S496c)

```
type context
contextDot : context * name -> path
```

```
datatype context
  = TOPLEVEL
  | INMODULE of path

fun contextDot (TOPLEVEL, name) = PNAME (genmodident name)
  | contextDot (INMODULE path, name) = PDOT (path, name)

fun contextString TOPLEVEL = "at top level"
  | contextString (INMODULE p) = "in module " ^ pathString p
```

Each definition appears in a known context and is typechecked using a given environment. (Some definitions, like a module-type definition for example, may appear only in a top-level context.) Typechecking a definition produces a named binding. is added to the environment by typdef.

S509c. ⟨*elaborate a Molecule definition* S507c⟩+≡ (S496c) ◁S509a

```
typdef : baredef * context * binding env -> (name * binding) list
```

```
fun typdef (d : baredef, context, Gamma) =
  let fun toplevel what =
        case context
          of TOPLEVEL => id
           | _ => raise TypeError (what ^ " cannot appear " ^
                                   contextString context)
      ⟨definition of mtypeof S513a⟩
  in  case d
        of EXP e =>
             let val what = "an expression (like " ^ expString e ^ ")"
             in  toplevel what (typdef (VAL ("it", e), context, Gamma))
             end
         | QNAME px =>
             let val what = "a qualified name (like " ^ pathexString px ^ ")"
             in  toplevel what (typdef (EXP (VAR px), context, Gamma))
             end
         ⟨named bindings for other forms of definition S510a⟩
  end
```

A module type is elaborated into internal form. Module types may appear only at top level.

S510a. ⟨*named bindings for other forms of definition* S510a⟩≡ (S509c) S510b ▷

```
  | MODULETYPE (T, mtx) =>
      let val mt = elabmt ((mtx, PNAME (MODTYPLACEHOLDER T)), Gamma)
      in  toplevel ("a module type (like " ^ T ^ ")")
          [(T, ENVMODTY mt)]
      end
```

A newly defined module is rooted at a location depending on its context, then typechecked with `mtypeof`.

S510b. ⟨*named bindings for other forms of definition* S510a⟩+≡ (S509c) ◁ S510a S510c ▷

```
  | MODULE (name, mx) =>
      let val root = contextDot (context, name)
          val mt   = mtypeof ((mx, root), Gamma)
      in  [(name, ENVMOD (mt, root))]
      end
```

When a generic-module definition is typechecked, each formal parameter adds to the environment in which subsequent formal parameters (and the body) are typechecked. A generic module may be defined only at top level.

S510c. ⟨*named bindings for other forms of definition* S510a⟩+≡ (S509c) ◁ S510b S510d ▷

```
  | GMODULE (f, formals, body) =>
      let val () = toplevel ("a generic module (like " ^ f ^ ")") ()
          val fpath     = contextDot (context, f)
          val idformals = map (fn (x, mtx) => (genmodident x, (x, mtx))) formals
          val resultpath = PAPPLY (fpath, map (PNAME o fst) idformals)

          fun addarg arg (args, res) = (arg :: args, res)

          fun arrowtype ((mid : modident, (x, mtx)) :: rest, Gamma) =
                let val mt = elabmt ((mtx, PNAME mid), Gamma)
                    val Gamma' = bind (x, ENVMOD (mt, PNAME mid), Gamma)
                in  addarg (mid, mt) (arrowtype (rest, Gamma'))
                end
            | arrowtype ([], Gamma) = ([], mtypeof ((body, resultpath), Gamma))
          val mt = MTARROW (arrowtype (idformals, Gamma))
      in  [(f, ENVMOD (mt, fpath))]
      end
```

As usual, `define` is syntactic sugar for a combination of `val-rec` and `lambda`.

S510d. ⟨*named bindings for other forms of definition* S510a⟩+≡ (S509c) ◁ S510c S510e ▷

```
  | DEFINE (name, tau, lambda as (formals, body)) =>
      let val funty = FUNTY (map (fn (n, ty) => ty) formals, tau)
      in  typdef (VALREC (name, funty, LAMBDA lambda), context, Gamma)
      end
```

The `val` and `val-rec` forms are typechecked as in Typed μScheme.

S510e. ⟨*named bindings for other forms of definition* S510a⟩+≡ (S509c) ◁ S510d S511a ▷

```
  | VAL (x, e) =>
      let val tau = typeof (e, Gamma)
      in  [(x, ENVVAL tau)]
      end
```

S511a. ⟨*named bindings for other forms of definition* S510a⟩+≡ (S509c) ◁S510e S511b▷

```
| VALREC (f, tau, e as LAMBDA _) =>
    let val tau    = elabty (tau, Gamma)
        val Gamma' = bind (f, ENVVAL tau, Gamma)
        val tau'   = typeof (e, Gamma')
    in  if not (eqType (tau, tau')) then
          raise TypeError ("identifier " ^ f ^
                           " is declared to have type " ^
                           typeString tau ^ " but has actual type " ^
                           typeString tau')
        else
          [(f, ENVVAL tau)]
    end
| VALREC (name, tau, _) =>
    raise TypeError ("(val-rec [" ^ name ^ " : " ^ tyexString tau ^
                     "] ...) must use (lambda ...) on right-hand side")
```

A manifest type definition is added to the environment.

S511b. ⟨*named bindings for other forms of definition* S510a⟩+≡ (S509c) ◁S511a S511c▷

```
| TYPE (t, tx) =>
    let val tau = elabty (tx, Gamma)
    in  [(t, ENVMANTY tau)]
    end
```

Each of the remaining forms (algebraic data types, overloading) is typechecked in its own function.

S511c. ⟨*named bindings for other forms of definition* S510a⟩+≡ (S509c) ◁S511b

```
| DATA dd => typeDataDef (dd, context, Gamma)
| OVERLOAD ovl => ⟨return bindings from overload list ovl S512c⟩
```

Typechecking datatype definitions

A data definition is checked more or less as in μML (Chapter 8).

S511d. ⟨*elaboration and evaluation of* data *definitions for Molecule* S511d⟩≡ (S496c) S523a▷

```
typeDataDef : data_def * context * binding env -> (name * binding) list
```

```
fun typeDataDef ((T, vcons), context, Gamma) =
  let val tau    = TYNAME (contextDot (context, T))
      val Gamma' = bind (T, ENVMANTY tau, Gamma)
      fun translateVcon (K, tx) =
            (K, elabty (tx, Gamma'))
            handle TypeError msg =>
              raise TypeError ("in type of value constructor " ^
                               K ^ ", " ^ msg)
      val Ktaus = map translateVcon vcons

      fun validate (K, FUNTY (_, result)) =
            if eqType (result, tau) then
              ()
            else
              ⟨result type of K should be tau but is result S512a⟩
        | validate (K, tau') =
            if eqType (tau', tau) then
              ()
            else
              ⟨type of K should be tau but is tau' S512b⟩
      val () = app validate Ktaus
  in  (T, ENVMANTY tau) :: map (fn (K, tau) => (K, ENVVAL tau)) Ktaus
  end
```

S512a. ⟨*result type of* `K` *should be* `tau` *but is* `result` S512a⟩≡ (S511d)

```
raise TypeError ("value constructor " ^ K ^ " should return " ^ typeString tau ^
                 ", but it returns type " ^ typeString result)
```

S512b. ⟨*type of* `K` *should be* `tau` *but is* `tau'` S512b⟩≡ (S511d)

```
raise TypeError ("value constructor " ^ K ^ " should have " ^ typeString tau ^
                ", but it has type " ^ typeString tau')
```

Computing `overload` *bindings*

Any path bound to a value that has a `firstArgType` can be overloaded. Its (short) name `x` and type `tau` are added to the list of types at which name `x` is overloaded (the `currentTypes`). (If `x` is not overloaded yet, start with a fresh, empty list.)

S512c. ⟨*return bindings from overload list* `ovl` S512c⟩≡ (S511c) S512d ▷

`overloadBinding : pathex * binding env -> name * binding`

```
let fun overloadBinding (p, Gamma) =
      let val (tau, first) =
            case pathfind (p, Gamma)
              of ENVVAL tau =>
                   (tau, okOrTypeError (firstArgType (pathexString p, tau)))
               | c => ⟨can't overload a c S481b⟩
          val x = plast p

          val currentTypes =
            (case find (x, Gamma)
               of ENVOVLN vals => vals
                | _ => []) handle NotFound _ => []
      in  (x, ENVOVLN (tau :: currentTypes))
      end
```

A sequence of paths is overloaded one at a time. Each addition extends the environment.

S512d. ⟨*return bindings from overload list* `ovl` S512c⟩+≡ (S511c) ◁ S512c

`overloadBindings : pathex list * binding env -> (name * binding) list`

```
    fun overloadBindings (ps, Gamma) =
      let fun add (bs', Gamma, []) = bs'
            | add (bs', Gamma, p :: ps) =
                let val b = overloadBinding (p, Gamma)
                in  add (b :: bs', Gamma <+> [b], ps)
                end
      in  add ([], Gamma, ps)
      end
in  overloadBindings (ovl, Gamma)
end
```

Typechecking module definitions

Each form of module definition is checked in its own way.

S513a. ⟨*definition of* mtypeof S513a⟩≡ (S509c)

```
mtypeof   : moddef rooted * binding env -> modty
sealed    : modtyex * modty -> modty
principal : def list -> modty
elabdefs  : def list * context * binding env -> (name * component) list
```

```
fun mtypeof ((m, path), Gamma) =
  let fun ty (MPATH p) =
                strengthen (findModule (p, Gamma), elabpath (p, Gamma))
        | ty (MPATHSEALED (mtx, p)) = sealed (mtx, ty (MPATH p))
        | ty (MUNSEALED defs)       = principal defs
        | ty (MSEALED (mtx, defs))  = sealed (mtx, principal defs)
      and sealed (sealingMtx, sealedMt) =
            let val sealingMt = elabmt ((sealingMtx, path), Gamma)
            in  case implements (path, sealedMt, sealingMt)
                  of OK () => sealingMt
                   | ERROR msg => raise TypeError msg
            end
      and principal ds = MTEXPORTS (elabdefs (ds, INMODULE path, Gamma))
      and elabdefs ([],    c, Gamma) = []
        | elabdefs ((loc, d) :: ds, c, Gamma) =
            let val bindings = atLoc loc typdef (d, c, Gamma)
                val comps'   = List.mapPartial asComponent bindings
                val Gamma'   = Gamma <+> bindings
                val comps''  = elabdefs (ds, c, Gamma')
                ⟨definition of asUnique S513b⟩
            in  List.mapPartial (asUnique comps'') comps' @ comps''
            end
  in  ty m
  end
```

Function asUnique ensures that no name is defined more than once—except for values. Redefining types or modules would not be sound, but redefining values is OK.

S513b. ⟨*definition of* asUnique S513b⟩≡ (S513a)

```
asUnique : component env -> name * component -> (name * component) option
```

```
fun asUnique following (x, c) =
  let val c' = find (x, following)
  in  case (c, c')
        of (COMPVAL _, COMPVAL _) => NONE (* repeated values are OK *)
         | _ => raise TypeError ("Redefinition of " ^ whatcomp c ^ " " ^ x ^
                                   " in module " ^ pathString path)
  end handle NotFound _ => SOME (x, c)
```

T.8.8 *Substitutions (boring)*

Substitutions are used to propagate information about manifest types. Everything you need to know about substitution appears in Chapter 6.

S513c. ⟨*substitutions for Molecule* S513c⟩≡ (S475 S496c) S513d ▷

```
type rootsubst = (modident * path) list
val idsubst = []
```

```
type rootsubst
idsubst : rootsubst
```

S513d. ⟨*substitutions for Molecule* S513c⟩+≡ (S475 S496c) ◁ S513c S514a ▷

```
infix 7 |-->
fun id |--> p = [(id, p)]
```

```
|--> : modident * path -> rootsubst
```

S514a. ⟨*substitutions for Molecule* S513c⟩+≡ (S475 S496c) ◁ S513d S514b ▷

```
type tysubst
associatedWith : path * tysubst -> ty option
hasKey : tysubst -> path -> bool
```

```
type tysubst =
  (path * ty) list
fun associatedWith (x, []) =
      NONE
  | associatedWith (x, (key, value) :: pairs) =
      if x = key then SOME value else associatedWith (x, pairs)

fun hasKey [] x = false
  | hasKey ((key, value) :: pairs) x = x = key orelse hasKey pairs x
```

S514b. ⟨*substitutions for Molecule* S513c⟩+≡ (S475 S496c) ◁ S514a S514c ▷

```
pathsubstRoot : rootsubst -> path -> path
```

```
fun pathsubstRoot theta =
  let fun subst (PNAME id) =
            (case List.find (fn (id', p') => id = id') theta
               of SOME (_, p) => p
                | NONE => PNAME id)
        | subst (PDOT (p, x)) = PDOT (subst p, x)
        | subst (PAPPLY (p, ps)) = PAPPLY (subst p, map subst ps)
  in  subst
  end
```

S514c. ⟨*substitutions for Molecule* S513c⟩+≡ (S475 S496c) ◁ S514b S514d ▷

```
tysubstRoot : rootsubst -> ty -> ty
```

```
fun tysubstRoot theta (TYNAME p)          = TYNAME (pathsubstRoot theta p)
  | tysubstRoot theta (FUNTY (args, res)) =
      FUNTY (map (tysubstRoot theta) args, tysubstRoot theta res)
  | tysubstRoot theta ANYTYPE = ANYTYPE
```

Functions dom and compose may be familiar from Chapter 7.

S514d. ⟨*substitutions for Molecule* S513c⟩+≡ (S475 S496c) ◁ S514c S514e ▷

```
dom     : rootsubst -> modident set
compose : rootsubst * rootsubst -> rootsubst
```

```
fun dom theta =
  map (fn (a, _) => a) theta
fun compose (theta2, theta1) =
  let val domain  = union (dom theta2, dom theta1)
      val replace = pathsubstRoot theta2 o pathsubstRoot theta1 o PNAME
  in  map (fn a => (a, replace a)) domain
  end
```

S514e. ⟨*substitutions for Molecule* S513c⟩+≡ (S475 S496c) ◁ S514d S515a ▷

```
mtsubstRoot   : rootsubst -> modty     -> modty
compsubstRoot : rootsubst -> component -> component
```

```
fun bsubstRoot s =
  map (fn (x, a) => (x, s a))

fun mtsubstRoot theta =
  let fun s (MTEXPORTS comps) = MTEXPORTS (bsubstRoot (compsubstRoot theta) comps)
        | s (MTALLOF mts)     = MTALLOF (map s mts)
        | s (MTARROW (args, res)) = MTARROW (bsubstRoot s args, s res)
  in  s
  end
and compsubstRoot theta =
  let fun s (COMPVAL t) = COMPVAL (tysubstRoot theta t)
        | s (COMPABSTY path) = COMPABSTY (pathsubstRoot theta path)
        | s (COMPMANTY t)  = COMPMANTY (tysubstRoot theta t)
        | s (COMPMOD mt)  = COMPMOD (mtsubstRoot theta mt)
  in  s
  end
```

S515a. ⟨*substitutions for Molecule* S513c⟩+≡ (S475 S496c) ◁S514e S515b▷

```
tysubstManifest : tysubst -> ty -> ty
```

```
fun tysubstManifest mantypes =
  let fun r (TYNAME path) = getOpt (associatedWith (path, mantypes), TYNAME path)
        | r (FUNTY (args, res)) = FUNTY (map r args, r res)
        | r (ANYTYPE) = ANYTYPE
  in  r
  end
```

S515b. ⟨*substitutions for Molecule* S513c⟩+≡ (S475 S496c) ◁S515a

```
mtsubstManifest : tysubst -> modty -> modty
```

```
fun mtsubstManifest mantypes mt =
  let val newty = tysubstManifest mantypes
      fun newmt (MTEXPORTS cs) =
            MTEXPORTS (map (fn (x, c) => (x, newcomp c)) cs)
        | newmt (MTALLOF mts) =
            MTALLOF (map newmt mts)  (* can't violate unmix invariant *)
        | newmt (MTARROW (args, result)) =
            MTARROW (map (fn (x, mt) => (x, newmt mt)) args, newmt result)
      and newcomp (COMPVAL tau) = COMPVAL (newty tau)
        | newcomp (COMPABSTY p) =
           (case associatedWith (p, mantypes)
              of SOME tau => COMPMANTY tau
               | NONE => COMPABSTY p)   (* used to be this on every path *)
        | newcomp (COMPMANTY tau) = COMPMANTY (newty tau)
        | newcomp (COMPMOD mt) = COMPMOD (newmt mt)
  in  newmt mt
  end
```

T.8.9 Elaboration of syntax into types

Paths, types, declarations, and module types all have two forms: syntactic and internal. Each syntactic form is translated into the corresponding internal form by an *elaboration* function.

S515c. ⟨*elaboration of Molecule type syntax into types* S515c⟩≡ (S475 S496c) S516a▷

```
elabpath : pathex * binding env -> path
```

```
fun elabpath (px, Gamma) =
  let fun elab (PAPPLY (f, args)) = PAPPLY (elab f, map elab args)
        | elab (PDOT (p, x)) = PDOT (elab p, x)
        | elab (PNAME (loc, m)) =
            let fun bad aThing =
                  raise TypeError ("I was expecting " ^ m ^ " to refer to " ^
                                   "a module, but at " ^ srclocString loc ^
                                   ", it's " ^ aThing)
            in  case find (m, Gamma)
                  of ENVMODTY _ => bad "a module type"
                   | ENVMOD (mt, p) => p
                   | c => bad (whatdec c)
            end
  in  elab px
  end
```

ANYTYPE	S477a
type binding	S478a
COMPABSTY	S477d
COMPMANTY	S477d
COMPMOD	S477d
type component	S477d
COMPVAL	S477d
type env	304
ENVMOD	S478a
ENVMODTY	S478a
find	305b
FUNTY	S477a
type modident	S476a
type modty	S477d
MTALLOF	S477d
MTARROW	S477d
MTEXPORTS	S477d
PAPPLY	S476b
type path	S476b
type pathex	S476b
PDOT	S476b
PNAME	S476b
srclocString	S233b
type ty	S477a
TYNAME	S477a
TypeError	S213c
union	S217b
whatdec	S496b

S516a. ⟨*elaboration of Molecule type syntax into types* S515c⟩+≡ (S475 S496c) ◁S515c S516b▷

```
elabty : tyex * binding env -> ty
```

```
fun elabty (t, Gamma) =
  let fun elab (TYNAME px) =
            (case pathfind (px, Gamma)
               of ENVMANTY tau => tau
                | dec => raise TypeError ("I was expecting a type, but " ^
                                          pathexString px ^ " is " ^ whatdec dec))
        | elab (FUNTY (args, res)) = FUNTY (map elab args, elab res)
        | elab ANYTYPE = ANYTYPE
  in  elab t
  end
```

When a module type is elaborated, it's easiest to separate out the code that looks up named module types.

S516b. ⟨*elaboration of Molecule type syntax into types* S515c⟩+≡ (S475 S496c) ◁S516a S516c▷

```
findModty : name * binding env -> modty
```

```
fun findModty (x, Gamma) =
  case find (x, Gamma)
    of ENVMODTY mt => mt
     | dec => raise TypeError ("Tried to use " ^ whatdec dec ^ " " ^ x ^
                               " as a module type")
```

S516c. ⟨*elaboration of Molecule type syntax into types* S515c⟩+≡ (S475 S496c) ◁S516b S517b▷

```
elabmt : modtyx rooted * binding env -> modty
export : (name * decl) * ((name * component) list * binding env)
                         -> ((name * component) list * binding env)
```

```
fun elabmt ((mtx : modtyx, path), Gamma) =
  let fun elab (MTNAMEDX t) =
            mtsubstRoot (MODTYPLACEHOLDER t |--> path) (findModty (t, Gamma))
        | elab (MTEXPORTSX exports) =
            let val (this', _) = foldl (leftLocated export) ([], Gamma) exports
            in  MTEXPORTS (rev this')
            end
        | elab (MTALLOFX mts) =
            allofAt (map (located elab) mts, path)
        | elab (MTARROWX (args, body)) =
            let val resultName = PNAME (MODTYPLACEHOLDER "functor result")
                fun arrow ([], (loc, body), Gamma : binding env, idents') =
                      let val resultName = PAPPLY (path, reverse idents')
                      in
                      ([], atLoc loc elabmt ((body, resultName), Gamma))
                      end
                  | arrow (((mloc, m), (mtloc, mtx)) :: rest, body, Gamma, idents') =
                      let val modid = genmodident m
                          val modty = atLoc mtloc elabmt ((mtx, PNAME modid), Gamma)
                          val () = ⟨if modty is generic, bleat about m S517a⟩
                          val Gamma' = bind (m, ENVMOD (modty, PNAME modid), Gamma)
                          val (rest', body') =
                            arrow (rest, body, Gamma', PNAME modid :: idents')
                      in  ((modid, modty) :: rest', body')
                      end
            in  MTARROW (arrow (args, body, Gamma, []))
            end
```

S517a. ⟨*if* modty *is generic, bleat about* m S517a⟩≡ (S516c)

```
case modty
  of MTARROW _ =>
    raise TypeError
            ("module parameter " ^ m ^ " is generic, and a generic " ^
             "module may not take another generic module as a parameter")
   | _ => ()
```

S517b. ⟨*elaboration of Molecule type syntax into types* S515c⟩+≡ (S475 S496c) ◁S516c S517c▷

```
      export : (name * decl) * ((name * component) list * binding env)
                         -> ((name * component) list * binding env)

      and export ((x, ctx : decl), (theseDecls, Gamma)) =
            if isbound (x, theseDecls) then
              raise TypeError ("duplicate declaration of " ^ x ^
                               " in module type")
            else
              let val c = elabComp ((ctx, PDOT (path, x)), Gamma)
              in  ((x, c) :: theseDecls, bind (x, asBinding (c, path), Gamma))
              end
  in  elab mtx
  end
```

A declaration is elaborated into a component.

S517c. ⟨*elaboration of Molecule type syntax into types* S515c⟩+≡ (S475 S496c) ◁S517b S517d▷

```
                      elabComp : decl rooted * binding env -> component

and elabComp ((comp : decl, path), Gamma : binding env) : component =
  let fun ty t = elabty (t, Gamma)
  in  case comp
        of DECVAL tau  => COMPVAL (ty tau)
         | DECABSTY    => COMPABSTY path
         | DECMANTY t  => COMPMANTY (ty t)
         | DECMOD mt   => COMPMOD (elabmt ((mt, path), Gamma))
         | DECMODTY mt =>
             raise TypeError ("module type " ^ pathString path ^
                              " may not be a component of another module")
  end
```

I redefine elabmt to check for bugs in my type checker.

S517d. ⟨*elaboration of Molecule type syntax into types* S515c⟩+≡ (S475 S496c) ◁S517c

```
val elabmt = fn a =>
  let val mt = elabmt a
  in  if mixedManifestations mt then
        raise BugInTypeChecking
                ("invariant violation (mixed M): " ^ mtString mt)
      else
        mt
  end
```

T.8.10 Overloading support

A qualified name is overloaded based on the last part of the name.

S517e. ⟨*support for operator overloading in Molecule* S517e⟩≡ (S475) S518a▷

```
                                              plast : pathex -> name
fun plast (PDOT (_, x)) = x
  | plast (PNAME (_, x)) = x
  | plast (PAPPLY _) = "??last??"
```

Each APPLY node in the abstract-syntax tree stores the index of the overloaded instance to be used at that APPLY node. If the function applied is *not* an overloaded name, that index is −1.

S518a. ⟨*support for operator overloading in Molecule* S517e⟩+≡ (S475) ◁S517e

```
val notOverloadedIndex = ~1
```

T.9 Evaluation

The components of the evaluator and read-eval-print loop are organized as follows:

S518b. ⟨*evaluation, testing, and the read-eval-print loop for Molecule* S518b⟩≡ (S475)

```
fun basename (PDOT (_, x)) = PNAME x
  | basename (PNAME x) = PNAME x
  | basename (instance as PAPPLY _) = instance
⟨definitions of matchRef and Doesn'tMatch (from chunk 697b)⟩
⟨definitions of eval and evaldef for Molecule S518c⟩
⟨definitions of basis and processDef for Molecule S480b⟩

⟨shared definition of withHandlers S239a⟩
⟨shared unit-testing utilities S225a⟩
⟨definition of testIsGood for Molecule S541c⟩
⟨shared definition of processTests S226⟩
⟨shared read-eval-print loop S237⟩
```

The basename function defined here is used in Chapter 8 to match a value constructor against a value-constructor pattern. This computation makes it possible to match two references to the same value constructor even if the two references refer to different modules. (The fact that such references denote the same value constructor is guaranteed by the type system.)

T.9.1 Evaluating paths

Since a path may include an instantiation of a generic module, evalpath includes a function to apply the generic module to its arguments.

S518c. ⟨*definitions of* eval *and* evaldef *for Molecule* S518c⟩≡ (S518b) S519a ▷

```
evalpath : pathex * value ref env -> value
apply    : value  * value list    -> value
```

```
fun evalpath (p : pathex, rho) =
  let fun findpath (PNAME (srcloc, x)) = !(find (x, rho))
        | findpath (PDOT (p, x)) =
            (case findpath p
               of MODVAL comps => (!(find (x, comps))
                                   handle NotFound x =>
                                     raise BugInTypeChecking "missing component")
                | _ => raise BugInTypeChecking "selection from non-module")
        | findpath (PAPPLY (f, args)) = apply (findpath f, map findpath args)
  in  findpath p
  end
and apply (PRIMITIVE prim, vs) = prim vs
  | apply (CLOSURE ((formals, body), rho_c), vs) =
      (eval (body, bindList (formals, map ref vs, rho_c))
       handle BindListLength =>
         raise BugInTypeChecking ("Wrong number of arguments to closure; " ^
                                  "expected (" ^ spaceSep formals ^ ")"))
  | apply _ = raise BugInTypeChecking "applied non-function"
```

T.9.2 Evaluating expressions

Most of the implementation of the evaluator is almost identical to the implementation in Chapter 5. But there are some significant differences:

- Evaluation of APPLY checks to see if an overloaded name is being applied, and if so, selects its value from an array.
- Before LAMBDA can be turned into CLOSURE, its types have to be erased.
- The evaluator needs cases for new forms like TYAPPLY and TYLAMBDA (Chapter 6), VCONX and CASE (Chapter 8), and MODEXP (a module-valued expression).

And as in Chapters 6 and 8, many potential run-time errors should be impossible because the relevant code would be rejected by the type checker. If one of those errors occurs anyway, the evaluator raises the exception BugInTypeChecking, not RuntimeError as in Chapter 5.

S519a. ⟨*definitions of* eval *and* evaldef *for Molecule* S518c⟩+≡ (S518b) ◁S518c S521b▷

```
eval : exp * value ref env -> value
ev   : exp                 -> value
```

```
  and eval (e, rho : value ref env) =
    let fun ev (LITERAL n) = n
          ⟨more alternatives for ev for Molecule S519b⟩
          | ev (EXP_AT (loc, e)) = atLoc loc ev e
    in  ev e
    end
```

Code for variables is just as in Chapter 5.

S519b. ⟨*more alternatives for* ev *for Molecule* S519b⟩≡ (S519a) S519c▷

```
  | ev (VAR p) = evalpath (p, rho)
  | ev (SET (n, e)) =
      let val v = ev e
      in  find (n, rho) := v;
          unitVal
      end
```

S519c. ⟨*more alternatives for* ev *for Molecule* S519b⟩+≡ (S519a) ◁S519b S519d▷

```
  | ev (VCONX c) = evalpath (pathexOfVcon c, rho)
  | ev (CASE (LITERAL v, (p, e) :: choices)) =
      (let val rho' = matchRef (p, v)
       in  eval (e, rho <+> rho')
       end
       handle Doesn'tMatch => ev (CASE (LITERAL v, choices)))
  | ev (CASE (LITERAL v, [])) =
      raise RuntimeError ("'case' does not match " ^ valueString v)
  | ev (CASE (e, choices)) =
      ev (CASE (LITERAL (ev e), choices))
```

Code for control flow is just as in Chapter 5.

S519d. ⟨*more alternatives for* ev *for Molecule* S519b⟩+≡ (S519a) ◁S519c S520a▷

```
  | ev (IFX (e1, e2, e3)) = ev (if projectBool (ev e1) then e2 else e3)
  | ev (WHILEX (guard, body)) =
      if projectBool (ev guard) then
        (ev body; ev (WHILEX (guard, body)))
      else
        unitVal
  | ev (BEGIN es) =
      let fun b (e::es, lastval) = b (es, ev e)
            | b (   [], lastval) = lastval
      in  b (es, unitVal)
      end
```

Code for a `lambda` removes the types from the abstract syntax.

S520a. ⟨*more alternatives for* `ev` *for Molecule* S519b⟩+≡ (S519a) ◁ S519d S520b ▷

```
  | ev (LAMBDA (args, body)) = CLOSURE ((map (fn (x, ty) => x) args, body), rho)
```

Each `APPLY` node contains a reference to an integer `i` that is set by the type checker. If `i` is negative, then what's being applied is *not* an overloaded name, and the value being applied is computed by `ev` as usual. If `i` is nonnegative, then name `f` evaluates to an array of possible functions, from which the correct function is found at index `i`. Once the function to be applied (`fv`) has been computed, it is applied just as in Chapter 5.

S520b. ⟨*more alternatives for* `ev` *for Molecule* S519b⟩+≡ (S519a) ◁ S520a S520c ▷

```
  | ev (APPLY (f, args, ref i))  =
     let val fv =
           if i < 0 then
             ev f
           else
             case ev f
               of ARRAY a =>
                    (Array.sub (a, i)
                     handle Subscript =>
                       raise BugInTypeChecking "overloaded index")
                | _ => raise BugInTypeChecking "overloaded name is not array"
     in  case fv
           of PRIMITIVE prim => prim (map ev args)
            | CLOSURE clo => ⟨apply closure clo to args 310c⟩
            | v => raise BugInTypeChecking "applied non-function"
     end
```

Code for the `LETX` family is as in Chapter 5.

S520c. ⟨*more alternatives for* `ev` *for Molecule* S519b⟩+≡ (S519a) ◁ S520b S520d ▷

```
  | ev (LETX (LET, bs, body)) =
      let val (names, values) = ListPair.unzip bs
      in  eval (body, bindList (names, map (ref o ev) values, rho))
      end
  | ev (LETX (LETSTAR, bs, body)) =
      let fun step ((x, e), rho) = bind (x, ref (eval (e, rho)), rho)
      in  eval (body, foldl step rho bs)
      end
```

S520d. ⟨*more alternatives for* `ev` *for Molecule* S519b⟩+≡ (S519a) ◁ S520c S520e ▷

```
  | ev (LETRECX (bs, body)) =
      let val (lhss, values) = ListPair.unzip bs
          val names = map fst lhss
          fun unspecified () = NUM 42
          val rho' = bindList (names, map (fn _ => ref (unspecified())) values, rho)
          val updates = map (fn (x, e) => (x, eval (e, rho'))) bs
      in  List.app (fn ((x, _), v) => find (x, rho') := v) updates;
          eval (body, rho')
      end
```

Evaluating a module expression produces a module value.

S520e. ⟨*more alternatives for* `ev` *for Molecule* S519b⟩+≡ (S519a) ◁ S520d S521a ▷

```
  | ev (MODEXP components) =
      let fun step ((x, e), (results', rho)) =
            let val loc = ref (eval (e, rho))
            in  ((x, loc) :: results', bind (x, loc, rho))
            end
          val (results', _) = foldl step ([], rho) components
      in  MODVAL results'
      end
```

S521a. ⟨*more alternatives for* `ev` *for Molecule* S519b⟩+≡ (S519a) ◁ S520e

```
| ev (ERRORX es) =
    raise RuntimeError (spaceSep (map (valueString o ev) es))
```

T.9.3 Evaluating definitions

Definitions may appear either at top level or inside a module. Function `evaldef` is called only for a definition that appears at top level. It returns the new environment and the list of values defined. (A typical definition defines just one value, but a `data` definition may define many values, and a `type` definition doesn't define any.) The real work of evaluating definitions is done by `defbindings`; it computes a list of bindings that may either be added to an environment (when the definition appears at top level) or may be used to form a module value (when the definition appears inside a module).

S521b. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S518c⟩+≡ (S518b) ◁ S519a S521c ▷

```
                evaldef : baredef * value ref env -> value ref env * value list
fun evaldef (d, rho) =
  let val bindings = defbindings (d, rho)
  in  (rho <+> bindings, map (! o snd) bindings)
  end
```

Function `defbindings` allocates and initializes a new mutable reference cell for each name introduced by the definition. The evaluation of the classic four definition forms from Chapter 2 proceeds just as in Chapter 2 (or Chapter 5). Type soundness requires a change in the evaluation rule for `VAL`; as described in Exercise 46 in Chapter 2, `VAL` must always create a new binding.

S521c. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S518c⟩+≡ (S518b) ◁ S521b S521d ▷

```
               defbindings : baredef * value ref env -> (name * value ref) list
and defbindings (VAL (x, e), rho) =
      [(x, ref (eval (e, rho)))]
  | defbindings (VALREC (x, tau, e), rho) =
      let val this = ref (SYM "placedholder for val rec")
          val rho' = bind (x, this, rho)
          val v    = eval (e, rho')
          val _    = this := v
      in  [(x, this)]
      end
  | defbindings (EXP e, rho) =
      defbindings (VAL ("it", e), rho)
  | defbindings (DEFINE (f, tau, lambda), rho) =
      defbindings (VALREC (f, tau, LAMBDA lambda), rho)
```

In the `VALREC` case, the interpreter evaluates `e` while `name` is still bound to `NIL`—that is, before the assignment to `find (name, rho)`. Therefore, as in Typed μScheme, evaluating `e` must not evaluate `name`—because the mutable cell for `name` does not yet contain its correct value.

Evaluating a qualified name doesn't bind it.

S521d. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S518c⟩+≡ (S518b) ◁ S521c S522a ▷

```
  | defbindings (QNAME _, rho) =
      []
```

The definition of a manifest type or a module type doesn't introduce any value at all.

S522a. ⟨*definitions of* eval *and* evaldef *for Molecule* S518c⟩+≡ (S518b) ◁S521d S522b▷

```
  | defbindings (TYPE _, _) =
    []
  | defbindings (MODULETYPE (a, _), rho) =
    []
```

Evaluating a data definition produces one value for each value constructor. If the type of the constructor is a function type, the value constructor is represented by a function; otherwise it's represented by a constructed value.

S522b. ⟨*definitions of* eval *and* evaldef *for Molecule* S518c⟩+≡ (S518b) ◁S522a S522c▷

```
  | defbindings (DATA (t, typed_vcons), rho) =
    let fun binding (K, tau) =
          let val v =
                case tau
                 of FUNTY _ => PRIMITIVE (fn vs => CONVAL (PNAME K, map ref vs))
                  | _ => CONVAL (PNAME K, [])
          in  (K, ref v)
          end
    in  map binding typed_vcons
    end
```

An exporting module is evaluated below by evalmod, which is defined below. A generic module evaluates to a closure; the body of the closure is an expression constructed by modexp, which is also defined below.

S522c. ⟨*definitions of* eval *and* evaldef *for Molecule* S518c⟩+≡ (S518b) ◁S522b S522d▷

```
  | defbindings (MODULE (x, m), rho) =
    [(x, ref (evalmod (m, rho)))]
  | defbindings (GMODULE (f, formals, body), rho) =
    [(f, ref (CLOSURE ((map fst formals, modexp body), rho)))]
```

A declaration of an overloaded name evaluates to an array, as used in the case for evaluating an APPLY expression above.

S522d. ⟨*definitions of* eval *and* evaldef *for Molecule* S518c⟩+≡ (S518b) ◁S522c S522e▷

```
  | defbindings (OVERLOAD ps, rho) =
    let fun overload (p :: ps, rho) =
              let val x = plast p
                  val v = extendOverloadTable (x, evalpath (p, rho), rho)
                  val loc = ref (ARRAY v)
              in  (x, loc) :: overload (ps, bind (x, loc, rho))
              end
          | overload ([], rho) = []
    in  overload (ps, rho)
    end
```

An array is created by extending an existing array, or if no array exists yet, by extending the empty array.

S522e. ⟨*definitions of* eval *and* evaldef *for Molecule* S518c⟩+≡ (S518b) ◁S522d S523b▷

```
      extendOverloadTable : name * value * value ref env -> value array
 and extendOverloadTable (x, v, rho) =
  let val currentVals =
        (case find (x, rho)
          of ref (ARRAY a) => a
           | _ => Array.fromList [])
        handle NotFound _ => Array.fromList []
  in  Array.tabulate (1 + Array.length currentVals,
                      fn 0 => v | i => Array.sub (currentVals, i - 1))
  end
```

As in Chapter 8, a `data` definition is evaluated separately from other forms. Function `evalDataDef` returns not the values of the new value constructors, but their names.

S523a. ⟨*elaboration and evaluation of* `data` *definitions for Molecule* S511d⟩+≡ (S496c) ◁S511d

```
                evalDataDef : data_def * value ref env -> value ref env * string list

fun evalDataDef ((_, typed_vcons), rho) =
  let fun addVcon ((K, t), rho) =
        let val v = if isfuntype t then
                      PRIMITIVE (fn vs => CONVAL (PNAME K, map ref vs))
                    else
                      CONVAL (PNAME K, [])
        in  bind (K, ref v, rho)
        end
  in  (foldl addVcon rho typed_vcons, map fst typed_vcons)
  end
```

T.9.4 Evaluating modules

An *exporting* module is evaluated by evaluating whatever form defines it: either a path or a list of definitions.

S523b. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S518c⟩+≡ (S518b) ◁S522e S523c▷

```
                                        evalmod : moddef * value ref env -> value

and evalmod (MSEALED (_, ds), rho) = evalmod (MUNSEALED ds, rho)
  | evalmod (MPATH p, rho) = evalpath (p, rho)
  | evalmod (MPATHSEALED (mtx, p), rho) = evalpath (p, rho)
  | evalmod (MUNSEALED defs, rho) = MODVAL (rev (defsbindings (defs, rho)))
```

A list of definitions is evaluated by calling `defbindings` on each definition. Earlier definitions add to the environment used to evaluate later definitions. Therefore, bindings are added using `foldl`; I can't use `<+>` here.

S523c. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S518c⟩+≡ (S518b) ◁S523b S523d▷

```
             defsbindings : def list * value ref env -> (name * value ref) list

and defsbindings ([],   rho) = []
  | defsbindings (d::ds, rho) =
      let val bs   = leftLocated defbindings (d, rho)
          val rho' = foldl (fn ((x, loc), rho) => bind (x, loc, rho)) rho bs
      in  bs @ defsbindings (ds, rho')
      end
```

A *generic* module is evaluated by creating a closure. The body of that closure is an expression constructed by `modexp`. For the common case in which the body is defined by a list of definitions, those definitions are converted into named expressions by `defexp`.

S523d. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S518c⟩+≡ (S518b) ◁S523c S524a▷

```
                                                       modexp : moddef -> exp

and modexp (MPATH px)             = VAR px
  | modexp (MPATHSEALED (_, px)) = VAR px
  | modexp (MSEALED (_, defs))   = MODEXP (concatMap (located defexps) defs)
  | modexp (MUNSEALED defs)      = MODEXP (concatMap (located defexps) defs)
```

`ARRAY`	S479b
`bind`	305d
`CLOSURE`	S479b
`concatMap`	S475
`CONVAL`	S479b
`DATA`	S479c
`defbindings`	S521c
`defexps`	S524a
type `env`	304
`evalpath`	S518c
`find`	305b
`fst`	S249b
`FUNTY`	S477a
`GMODULE`	S479c
`isfuntype`	S477c
`leftLocated`	S234f
`located`	S234f
type `moddef`	S479c
`MODEXP`	S479a
`MODULE`	S479c
`MODULETYPE`	S479c
`MODVAL`	S479b
`MPATH`	S479c
`MPATHSEALED`	S479c
`MSEALED`	S479c
`MUNSEALED`	S479c
type `name`	303
`NotFound`	305b
`OVERLOAD`	S479c
`plast`	S517e
`PNAME`	S476b
`PRIMITIVE`	S479b
`TYPE`	S479c
`VAR`	S479a

Function `defexps` has a lot in common with `defbindings`. I would like to combine them, but I have not figured out how to do so. (Shipping is also a feature.)

S524a. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S518c⟩+≡ (S518b) ◁S523d

```
defexps : baredef -> (name * exp) list
```

```
and defexps (VAL (x, e)) = [(x, e)]
  | defexps (VALREC (x, tau, e)) =
      let val nullsrc : srcloc = ("translated name in VALREC", ~1)
      in  [(x, LETRECX ([((x, tau), e)], VAR (PNAME (nullsrc, x))))]
      end
  | defexps (EXP e) =  [("it", e)]
  | defexps (QNAME _) = []
  | defexps (DEFINE (f, tau, lambda)) = defexps (VALREC (f, tau, LAMBDA lambda))
  | defexps (TYPE _) = []
  | defexps (DATA (t, typed_vcons)) =
      let fun vconExp (K, t) =
            let val v = if isfuntype t then
                          PRIMITIVE (fn vs => CONVAL (PNAME K, map ref vs))
                        else
                          CONVAL (PNAME K, [])
            in  (K, LITERAL v)
            end
      in  map vconExp typed_vcons
      end
  | defexps (MODULE (x, m)) = [(x, modexp m)]
  | defexps (GMODULE (f, formals, body)) =
      [(f, LAMBDA (map (fn (x, _) => (x, ANYTYPE)) formals, modexp body))]
  | defexps (MODULETYPE (a, _)) = []
  | defexps (OVERLOAD ovls) = unimp "overloadiang within generic module"
```

T.10 String conversion

String conversion in Molecule differs from string conversion in other languages in two ways:

- There are many more types of things to be converted.
- Some things, notably types, are converted to a value of type `doc`, so they can be prettyprinted.

T.10.1 Conversion for messaging about the environment

S524b. ⟨*environments for Molecule's defined names* S496a⟩+≡ (S475) ◁S496b

```
compString : binding -> string
```

```
fun compString (ENVVAL tau) = "a value of type " ^ typeString tau
  | compString (ENVMANTY tau) = "manifest type " ^ typeString tau
  | compString (ENVOVLN _) = "an overloaded name"
  | compString (ENVMOD (mt, path)) = "module " ^ pathString path ^
                                     " of type " ^ mtString mt
  | compString (ENVMODTY _) = "a module type"
```

T.10.2 Conversion of values

The conversion of constructed values is very close to what's in μML (Appendix S), but I have not tried to share code.

S525a. ⟨*string conversion of Molecule values* S525a⟩≡ (S478c) S525b ▷

`vconString : vcon -> string`

```
fun vconString (PNAME c) = c
  | vconString (PDOT (m, c)) = vconString m ^ "." ^ c
  | vconString (PAPPLY _) = "can't happen! (vcon PAPPLY)"
```

S525b. ⟨*string conversion of Molecule values* S525a⟩+≡ (S478c) ◁ S525a S525c ▷

`valueString : value -> string`

```
fun valueString (CONVAL (PNAME "cons", [ref v, ref vs])) = consString (v, vs)
  | valueString (CONVAL (PNAME "'()",  []))      = "()"
  | valueString (CONVAL (c, []))  = vconString c
  | valueString (CONVAL (c, vs))  =
      "(" ^ vconString c ^ " " ^ spaceSep (map (valueString o !) vs) ^ ")"
  | valueString (NUM n      )    =
      String.map (fn #"~" => #"-" | c => c) (Int.toString n)
  | valueString (SYM v      )    = v
  | valueString (CLOSURE   _)    = "<function>"
  | valueString (PRIMITIVE _)    = "<function>"
  | valueString (MODVAL _)       = "<module>"
  | valueString (ARRAY a)        =
      "[" ^ spaceSep (map valueString (Array.foldr op :: [] a)) ^ "]"
```

Render a cons cell as a list.

S525c. ⟨*string conversion of Molecule values* S525a⟩+≡ (S478c) ◁ S525b

`consString : value * value -> string`

```
and consString (v, vs) =
      let fun tail (CONVAL (PNAME "cons", [ref v, ref vs])) =
                " " ^ valueString v ^ tail vs
            | tail (CONVAL (PNAME "'()", [])) =
                ")"
            | tail _ =
                raise BugInTypeChecking
                  "bad list constructor (or cons/'() redefined)"
      in  "(" ^ valueString v ^ tail vs
      end
```

T.10.3 Conversion of types and module types

Modules with the same name but different serial numbers must print differently.

S525d. ⟨*string conversion of Molecule types and module types* S525d⟩≡ (S478c) S525e ▷

`modidentString : modident -> string`

```
fun modidentString (MODCON { printName = m, serial = 0 }) = m
  | modidentString (MODCON { printName = m, serial = k }) =
      m ^ "@{" ^ intString k ^ "}"
  | modidentString (MODTYPLACEHOLDER s) = "<signature: " ^ s ^ ">"
```

S525e. ⟨*string conversion of Molecule types and module types* S525d⟩+≡ (S478c) ◁ S525d S526a ▷

`pathString : path -> string`

```
fun pathString (PNAME a) = modidentString a
  | pathString (PDOT (PNAME (MODTYPLACEHOLDER _), x)) = x
  | pathString (PDOT (p, x)) = pathString p ^ "." ^ x
  | pathString (PAPPLY (f, args)) =
      spaceSep ("(@m" :: pathString f :: map pathString args) ^ ")"
```

S526a. ⟨*string conversion of Molecule types and module types* S525d⟩+≡ (S478c) ◁ S525e S526b ▷

```
pathexString : pathex -> string
```

```
val pathexString =
  let fun s (PNAME a) = snd a
        | s (PDOT (p, x)) = s p ^ "." ^ x
        | s (PAPPLY (f, args)) = spaceSep ("(@m" :: s f :: map s args) ^ ")"
  in  s
  end
```

Function typeString' takes as argument function ps, which converts a path to a string.

S526b. ⟨*string conversion of Molecule types and module types* S525d⟩+≡ (S478c) ◁ S526a S526c ▷

```
tyexString : tyex -> string
typeString : ty -> string
```

```
fun typeString' ps (TYNAME p) = ps p
  | typeString' ps (FUNTY (args, res)) =
      let val ts = typeString' ps
      in  "(" ^ spaceSep (map ts args @ "->" :: [ts res]) ^ ")"
      end
  | typeString' ps ANYTYPE = "<any type>"

val typeString = typeString' pathString
val tyexString = typeString' pathexString
```

S526c. ⟨*string conversion of Molecule types and module types* S525d⟩+≡ (S478c) ◁ S526b S528a ▷

```
mtString : modty -> string
ncompString : name * component -> string
```

```
fun mtString (MTEXPORTS []) = "(exports)"
  | mtString (MTEXPORTS comps) =
      "(exports " ^ spaceSep (map ncompString comps) ^ ")"
  | mtString (MTALLOF  mts) = "(allof " ^ spaceSep (map mtString mts) ^ ")"
  | mtString (MTARROW (args, res)) =
      "(" ^ spaceSep (map modformalString args) ^ " --m-> " ^ mtString res ^ ")"
and modformalString (m, t) = "[" ^ modidentString m ^ " : " ^ mtString t ^ "]"
and ncompString (x, c) = (* "named component" *)
  case c
    of COMPVAL tau => "[" ^ x ^ " : " ^ typeString tau ^ "]"
     | COMPABSTY _    => "[abstype " ^ x ^ "]"
     | COMPMANTY tau => "[type " ^ x ^ " " ^ typeString tau ^ "]"
     | COMPMOD mt => "(module [" ^ x ^ " : " ^ mtString mt ^ "])"
```

T.10.4 Prettyprinting of types and module types

"String conversion" includes not only conversion to strings, but also conversion to docs (for prettyprinting). A couple of additional combinators supplement the ones from Appendix J.

S526d. ⟨*prettyprinting combinators* S526d⟩≡ (S475)

```
infix 2 ^/+
fun l ^/+ r = l ^^ indent (2, agrp (brk ^^ r))

fun separateDoc (zero, sep) =
  (* list with separator *)
  let fun s []     = zero
        | s [x]    = x
        | s (h::t) = h ^^ sep ^^ s t
  in  s
  end
val brkSep  = separateDoc (doc "", brk)
```

For arrow types (functions or generic modules).

S527a. ⟨*prettyprinting of Molecule types and module types* S527a⟩≡ (S478c) S527b ▷

`arrowdoc : doc list -> string -> doc -> doc`

```
fun arrowdoc [] arrow res =
      doc "( " ^^ doc arrow ^^ doc " " ^^
      indent(3 + size arrow, res) ^^ doc ")"
  | arrowdoc (arg :: args) arrow res =
      let val docs =  (doc "(" ^^ arg)
                   :: args
                   @  doc arrow
                   :: [res ^^ doc ")"]
      in  indent(1, agrp (brkSep docs))
      end
```

S527b. ⟨*prettyprinting of Molecule types and module types* S527a⟩+≡ (S478c) ◁ S527a S527c ▷

`typeDoc : ty -> doc`

```
fun typeDoc (TYNAME p) = doc (pathString p)
  | typeDoc (FUNTY (args, res)) =
      arrowdoc (map typeDoc args) "->" (typeDoc res)
  | typeDoc ANYTYPE = doc "<any-type>"
```

S527c. ⟨*prettyprinting of Molecule types and module types* S527a⟩+≡ (S478c) ◁ S527b S527d ▷

`mtDoc : modty -> doc`

```
fun stdindent doc = indent (2, doc)

fun mtDoc (MTEXPORTS []) = doc "(exports)"
  | mtDoc (MTEXPORTS comps) =
      agrp (doc "(exports" ^^
            stdindent (brk ^^ brkSep (map ncompDoc comps) ^^ doc ")"))
  | mtDoc (MTALLOF  mts) =
      doc "(allof" ^/+ brkSep (map mtDoc mts) ^^ doc ")"
  | mtDoc (MTARROW (args, res)) =
      arrowdoc (map modformalDoc args) "--m->" (mtDoc res)
```

S527d. ⟨*prettyprinting of Molecule types and module types* S527a⟩+≡ (S478c) ◁ S527c S527e ▷

`modformalDoc : modident * modty -> doc`
`ncompDoc : name * component -> doc`

```
and modformalDoc (m, t) =
    agrp (doc "[" ^^ doc (modidentString m) ^^ doc " :" ^^
          indent(2, brk ^^ mtDoc t ^^ doc "]"))
and ncompDoc (x, c) =
   case c
    of COMPVAL tau =>
         agrp (doc "[" ^^ doc x ^^ doc " :" ^^
               stdindent (brk ^^ typeDoc tau ^^ doc "]"))
     | COMPABSTY _   => doc ("[abstype " ^ x ^ "]")
     | COMPMANTY tau => agrp (doc "[type " ^^ doc x ^^ doc " " ^^
                                  indent (4, typeDoc tau) ^^ doc "]")
     | COMPMOD mt =>
         agrp (doc "(module" ^^
                 indent(2, brk ^^ doc "[" ^^ doc x ^^ doc " :" ^^
                 indent(2, brk ^^ agrp (mtDoc mt ^^ doc"])"))))
```

S527e. ⟨*prettyprinting of Molecule types and module types* S527a⟩+≡ (S478c) ◁ S527d

`ndecString : name * binding -> string`

```
fun ndecString (x, c) =
  case c
   of ENVVAL tau => "[" ^ x ^ " : " ^ typeString tau ^ "]"
    | ENVMANTY tau => "(type " ^ x ^ " " ^ typeString tau ^ ")"
    | ENVMOD (mt, _) => "(module [" ^ x ^ " : " ^ mtString mt ^ "])"
    | ENVOVLN _ => "<overloaded name " ^ x ^ " ...>"
    | ENVMODTY mt => "(module-type " ^ x ^ " " ^ mtString mt ^ ")"
```

S528a. ⟨*string conversion of Molecule types and module types* S525d⟩+≡ (S478c) ◁S526c S528b▷

```
mtxString : modtyex -> string
ncompxString : (name * decl) located -> string
```

```
fun mtxString (MTNAMEDX m) = m
  | mtxString (MTEXPORTSX []) = "(exports)"
  | mtxString (MTEXPORTSX lcomps) =
      "(exports " ^ spaceSep (map ncompxString lcomps) ^ ")"
  | mtxString (MTALLOFX  mts) =
      "(allof " ^ spaceSep (map (mtxString o snd) mts) ^ ")"
  | mtxString (MTARROWX (args, res)) =
      "(" ^ spaceSep (map modformalString args) ^ " --m-> " ^
            mtxString (snd res) ^ ")"
```

S528b. ⟨*string conversion of Molecule types and module types* S525d⟩+≡ (S478c) ◁S528a

```
and modformalString (m, t) = "[" ^ snd m ^ " : " ^ mtxString (snd t) ^ "]"
and ncompxString (loc, (x, c)) =
  case c
    of DECVAL tau => "[" ^ x ^ " : " ^ tyexString tau ^ "]"
     | DECABSTY   => "(abstype " ^ x ^ ")"
     | DECMANTY tau => "(type " ^ x ^ " " ^ tyexString tau ^ ")"
     | DECMOD mt => "(module [" ^ x ^ " : " ^ mtxString mt ^ "])"
     | DECMODTY mt => "(module-type " ^ x ^ " " ^ mtxString mt ^ ")"
```

T.10.5 Conversion of abstract syntax

S529. ⟨*string conversion of Molecule's abstract syntax* S529⟩≡ (S478c) S530a ▷

```
expString : exp -> string
```

```
fun stripExpAt (EXP_AT (_, e)) = stripExpAt e
  | stripExpAt e = e

fun expString e =
  let fun bracket s = "(" ^ s ^ ")"
      fun sqbracket s = "[" ^ s ^ "]"
      val bracketSpace = bracket o spaceSep
      fun exps es = map expString es
      fun withBindings (keyword, bs, e) =
        bracket (spaceSep [keyword, bindings bs, expString e])
      and bindings bs = bracket (spaceSep (map binding bs))
      and binding (x, e) = sqbracket (x ^ " " ^ expString e)
      fun formal (x, ty) = sqbracket (x ^ " : " ^ tyexString ty)
      fun tbindings bs = bracket (spaceSep (map tbinding bs))
      and tbinding ((x, tyex), e) =
        bracket (formal (x, tyex) ^ " " ^ expString e)
      val letkind = fn LET => "let" | LETSTAR => "let*"
  in  case e
        of LITERAL v => valueString v
         | VAR name => pathexString name
         | IFX (e1, e2, e3) => bracketSpace ("if" :: exps [e1, e2, e3])
         | SET (x, e) => bracketSpace ["set", x, expString e]
         | WHILEX (c, b) =>
             bracketSpace ["while", expString c, expString b]
         | BEGIN es => bracketSpace ("begin" :: exps es)
         | APPLY (e, es, _) => bracketSpace (exps (e::es))
         | LETX (lk, bs, e) =>
             bracketSpace [letkind lk, bindings bs, expString e]
         | LETRECX (bs, e) =>
             bracketSpace ["letrec", tbindings bs, expString e]
         | LAMBDA (xs, body) =>
             bracketSpace ("lambda" :: map formal xs @ [expString body])
         | VCONX vcon => vconString vcon
         | CASE (e, matches) =>
             let fun matchString (pat, e) =
                   sqbracket (spaceSep [patString pat, expString e])
             in  bracketSpace
                   ("case" :: expString e :: map matchString matches)
             end
         | MODEXP components =>
             bracketSpace ("modexp" :: map binding components)
         | ERRORX es => bracketSpace ("error" :: exps es)
         | EXP_AT (_, e) => expString e
  end
```

S530a. ⟨*string conversion of Molecule's abstract syntax* S529⟩+≡ (S478c) ◁S529

`defString : baredef -> string`

```
fun defString d =
  let fun bracket s = "(" ^ s ^ ")"
      val bracketSpace = bracket o spaceSep
      fun sq s = "[" ^ s ^ "]"
      val sqSpace = sq o spaceSep
      fun formal (x, t) = "[" ^ x ^ " : " ^ tyexString t ^ "]"
  in  case d
        of EXP e         => expString e
         | VAL    (x, e) => bracketSpace ["val",     x, expString e]
         | VALREC (x, t, e) =>
             bracketSpace ["val-rec", sqSpace [x, ":", tyexString t],
                                      expString e]
         | DEFINE (f, ty, (formals, body)) =>
             bracketSpace ["define", tyexString ty, f,
                                      bracketSpace (map formal formals), expString body]
         | QNAME p => pathexString p
         | TYPE (t, tau) => bracketSpace ["type", t, tyexString tau]
         | DATA (t, _) => bracketSpace ["data", t, "..."]
         | OVERLOAD paths => bracketSpace ("overload" :: map pathexString paths)
         | MODULE (m, _) => bracketSpace ["module", m, "..."]
         | GMODULE (m, _, _) => bracketSpace ["generic-module", m, "..."]
         | MODULETYPE (t, mt) => bracketSpace ["module-type", t, "..."]
  end
```

T.11 Lexical analysis

In order to recognize qualified names, Molecule needs its own lexer: one in which dots separate tokens. In this lexer, *every* name is recognized as `DOTTED`—even one with no dots. A name with no dots is represented as `DOTTED` with an empty list.

S530b. ⟨*lexical analysis for Molecule* S530b⟩≡ (S532b) S530c ▷

```
datatype pretoken
  = QUOTE
  | INT      of int
  | RESERVED of string
  | DOTTED   of string * string list
  | DOTNAMES of string list (* .x.y and so on *)
type token = pretoken plus_brackets
```

S530c. ⟨*lexical analysis for Molecule* S530b⟩+≡ (S532b) ◁S530b S532a ▷

```
fun pretokenString (QUOTE)      = "'"
  | pretokenString (INT  n)     = intString n
  | pretokenString (DOTTED (s, ss))  = String.concatWith "." (s::ss)
  | pretokenString (DOTNAMES ss)= (concat o map (fn s => "." ^ s)) ss
  | pretokenString (RESERVED x) = x
val tokenString = plusBracketsString pretokenString
```

Molecule's reserved words are listed here, and if a single name x is reserved, it is converted from `DOTTED` (x, `[]`) to `RESERVED` x by function `reserve`.

S531a. ⟨*support for Molecule's reserved words* S531a⟩≡ (S532a)

```
val reserved =
  [ ⟨words reserved for Molecule types S533c⟩
  , ⟨words reserved for Molecule expressions S534e⟩
  , ⟨words reserved for Molecule definitions S539a⟩
  ]
fun isReserved x = member x reserved
fun reserve (token as DOTTED (s, [])) =
      if isReserved s then
        RESERVED s
      else
        token
  | reserve token = token
```

A dotted name is recognized by splitting the input characters into delimiters and nondelimiters. The delimiters include the usual delimiters from other lexers (brackets and so on, as defined by `isDelim`), plus the dot.

S531b. ⟨*lexing functions for Molecule's dotted names* S531b⟩≡ (S532a) S531c ▷

```
val isDelim = fn c => isDelim c orelse c = #"."
```

A dotted name is formed from a sequence of parts. Each part is either a contiguous sequence of nondelimiters or a single dot.

S531c. ⟨*lexing functions for Molecule's dotted names* S531b⟩+≡ (S532a) ◁ S531b S531d ▷

```
datatype part = DOT | NONDELIMS of string
val nondelims = (NONDELIMS o implode) <$> many1 (sat (not o isDelim) one)
val dot       = DOT <$ eqx #"." one
```

The sequence of parts is turned into a token by function `dottedNames`. Inner functions `preDot` and `postDot` are called before and after each dot, respectively. A token with consecutive dots or a final dot is ill formed.

S531d. ⟨*lexing functions for Molecule's dotted names* S531b⟩+≡ (S532a) ◁ S531c

`dottedNames : part list -> pretoken error`

```
fun dottedNames things =
  let fun preDot (ss', DOT :: things)    = postDot (ss', things)
        | preDot (ss', nil)              = OK (rev ss')
        | preDot (ss', NONDELIMS _ :: _) =
            raise InternalError "bad dot in lexer"
      and postDot (ss', DOT :: _) =
            ERROR "A qualified name may not contain consecutive dots"
        | postDot (ss', nil)      =
            ERROR "A qualified name may not end with a dot"
        | postDot (ss', NONDELIMS s :: things) =
            if isReserved s then
              ERROR ("reserved word '" ^ s ^ "' used in qualified name")
            else
              preDot (s :: ss', things)
  in  case things
        of NONDELIMS s :: things => preDot  ([], things) >>=+ curry DOTTED s
         | DOT         :: things => postDot ([], things) >>=+ DOTNAMES
         | [] => ERROR "Lexer is broken; report to nr@cs.tufts.edu"
  end
```

A token is either a quote mark; an integer literal; or a (dotted) name, possibly reserved. Or a bracket, which is taken care of by `bracketLexer`.

S532a. ⟨*lexical analysis for Molecule* S530b⟩+≡ (S532b) ◁ S530c

```
mclToken : token lexer
badReserved : name -> 'a error
```

```
local
  ⟨functions used in all lexers S384d⟩
  ⟨support for Molecule's reserved words S531a⟩
  ⟨lexing functions for Molecule's dotted names S531b⟩
in
  val mclToken =
    whitespace *>
    bracketLexer (  QUOTE   <$  eqx #"'" one
                <|> INT     <$> intToken isDelim
                <|> reserve <$> (dottedNames <$>! many1 (nondelims <|> dot))
                <|> noneIfLineEnds
                 )
  fun badReserved r =
    ERROR ("reserved word '" ^ r ^ "' where name was expected")
end
```

T.12 Parsing

Parsing code is shared with μML and Typed μScheme. And because those other parsers call `booltok` and `tyvar`, they have to be defined. Here they are defined as `pzero`, which always fails.

S532b. ⟨*lexical analysis and parsing for Molecule, providing* `filexdefs` *and* `stringsxdefs` S532b⟩≡ (S475)

```
⟨lexical analysis for Molecule S530b⟩
⟨parsers for Molecule tokens S532c⟩
val booltok = pzero
val tyvar = pzero : name parser
⟨parsers for μML value constructors and value variables (from chunk 697b)⟩
⟨parsers and parser builders for formal parameters and bindings S385c⟩
⟨parser builders for typed languages S401b⟩
⟨parsers and xdef streams for Molecule S532d⟩
⟨shared definitions of filexdefs and stringsxdefs S233a⟩
```

Parsers for tokens are defined as follows:

S532c. ⟨*parsers for Molecule tokens* S532c⟩≡ (S532b)

```
type 'a parser = (token, 'a) polyparser
val pretoken  = (fn (PRETOKEN t)=> SOME t  | _ => NONE) <$>? token : pretoken parser
val quote     = (fn (QUOTE)     => SOME () | _ => NONE) <$>? pretoken
val int       = (fn (INT   n)   => SOME n  | _ => NONE) <$>? pretoken
val namelike  = ((fn (DOTTED (x, []))   => SOME x  | _ => NONE) <$>? pretoken)
val dotted  = (fn (DOTTED (x, xs))   => SOME (x, xs)  | _ => NONE) <$>? pretoken
val dotnames = (fn (DOTNAMES xs)  => SOME xs | _ => NONE) <$>? pretoken
val reserved = (fn RESERVED r => SOME r | _ => NONE) <$>? pretoken
val name = asAscii namelike  (* reserved words handled in lexer *)



val arrow = eqx "->" reserved <|> eqx "--m->" reserved


val showErrorInput = (fn p => showErrorInput tokenString p)
```

To use the common `usageParsers` function, I need a recognizer for keywords. Which is useful for the language-specific parsers as well.

S532d. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩≡ (S532b) S533a ▷

```
fun kw keyword = eqx keyword reserved
fun usageParsers ps = anyParser (map (usageParser kw) ps)
```

A path may include an application to paths, which makes `path` recursive. And an application may be followed by more component selection (.x.y and so on), which is parsed by `dotnames`.

S533a. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S532d S533b▷

```
addDots : pathex -> name list -> pathex
path : pathex parser
```

```
fun addDots p xs =
  foldl (fn (x, p) => PDOT (p, x)) p xs
fun path tokens =
  let fun qname (loc, (x, xs)) = addDots (PNAME (loc, x)) xs
      val appl = curry PAPPLY <$> (PNAME <$> @@ name) <*> many path
  in  qname <$> @@ dotted
        <|>
      addDots <$> bracketKeyword (kw "@m", "(@m name path ...)", appl)
              <*> (dotnames <|> pure [])
  end tokens
```

A `tyex` is either a path or is a function type formed from types and an arrow. Function `br` handles the function type and any errors that may occur there. It also can provide a usage string.

S533b. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S533a S534a▷

```
mkTyex : (string * tyex parser -> tyex parser) -> tyex parser
tyex : tyex parser
```

```
fun mkTyex br tokens =
  let val ty = showErrorInput (mkTyex br)
      fun arrows []              [] = ERROR "empty type ()"
        | arrows (tycon::tyargs) [] = ERROR "missing @@ or ->"
        | arrows args            [rhs] =
            (case rhs of [result] => OK (FUNTY (args, result))
                       | [] => ERROR "no result type after function arrow"
                       | _  => ERROR ("multiple result types " ^
                                      "after function arrow"))
        | arrows args (_::_::_) = ERROR "multiple arrows in function type"
  in
      TYNAME <$> path
        <|>
      br ( "(ty ty ... -> ty)"
         ,  arrows <$> many ty <*>! many (kw "->" *> many ty)
         )
  end tokens
val tyex = mkTyex (showErrorInput o bracket)
```

The arrow is reserved for function types. The colon is reserved to mark formal parameters, and to improve error detection, I lump it in with types.

S533c. ⟨*words reserved for Molecule types* S533c⟩≡ (S531a S536a)

```
"->", ":"
```

A value constructor is like a path, except the last component must satisfy `isVcon`.

S534a. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S533b S534b▷

`vcon : vcon parser`

```
val bare_vcon = vcon
fun dottedVcon (x, xs) = addDots (PNAME x) xs
fun vconLast (PDOT (_, x)) = x
  | vconLast (PNAME x) = x
  | vconLast (PAPPLY _) = raise InternalError "application vcon"

val vcon =
  let fun notAVcon (loc, (x, xs)) =
        let val name = String.concatWith "." (x::xs)
        in  synerrorAt ("Expected value constructor, but got name " ^ name) loc
        end
  in  sat (isVcon o vconLast) (dottedVcon <$> dotted)
        <|> PNAME <$> bare_vcon
        <|> notAVcon <$>! @@ dotted
  end
```

Within a pattern, only a `vcon` may be applied.

S534b. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S534a S534c▷

`pattern : pat parser`

```
fun pattern tokens =  (
                WILDCARD    <$  eqx "_" vvar
      <|>       PVAR        <$> vvar
      <|> curry CONPAT      <$> vcon <*> pure []
      <|> bracket ( "(C x1 x2 ...) in pattern"
                  , curry CONPAT <$> vcon <*> many pattern
                  )
       ) tokens
```

Quoted literals need their own parser.

S534c. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S534b S534d▷

```
quoteName : string -> value
quotelit : value parser
```

```
fun quoteName "#f" = CONVAL (PNAME "#f", [])
  | quoteName "#t" = CONVAL (PNAME "#t", [])
  | quoteName s    = SYM s

fun quotelit tokens = (
         quoteName <$> name
    <|>  NUM <$> int
    <|>  (ARRAY o Array.fromList) <$> bracket ("(literal ...)", many quotelit)
    ) tokens
```

Molecule has its share of atomic expressions.

S534d. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S534c S535a▷

```
val atomicExp =  VAR <$> path
             <|> badReserved <$>! reserved
             <|> dotnames <!> "a qualified name may not begin with a dot"
             <|> LITERAL <$> NUM <$> int
             <|> VCONX <$> vcon
             <|> quote *> (LITERAL <$> quotelit)
```

For the non-atomic expressions, the following words are reserved:

S534e. ⟨*words reserved for Molecule expressions* S534e⟩≡ (S531a)

```
"@m", "if", "&&", "||", "set", "let", "let*", "letrec", "case", "lambda",
"val", "set", "while", "begin", "error",
"when", "unless", "assert"
```

Parsers for bindings and formal parameters are built as follows:

S535a. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S534d S535b▷

```
lformals : (name * tyex) list parser
```

```
fun bindTo exp = bracket ("[x e]", pair <$> name <*> exp)
val formal = bracket ("[x : ty]", pair <$> name <* kw ":" <*> tyex)
val lformals = bracket ("([x : ty] ...)", many formal)
fun nodupsty what (loc, xts) =
  nodups what (loc, map fst xts) >>=+ (fn _ => xts)
                                        (* error on duplicate names *)
```

In common with *μ*Smalltalk, Molecule allows a function body to contain a sequence of expressions; `begin` can be implicit. A `BEGIN` is added only when needed, by a "smart constructor."

S535b. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S535a S535c▷

```
smartBegin : exp list -> exp
```

```
fun smartBegin [e] = e
  | smartBegin es = BEGIN es
```

Short-circuit Boolean connectives desugar to `IFX` as follows:

S535c. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S535b S535d▷

```
cand : exp list -> exp
cor  : exp list -> exp
```

```
fun cand [e] = e
  | cand (e::es) = IFX (e, cand es, LITERAL (embedBool false))
  | cand [] = raise InternalError "parsing &&"

fun cor [e] = e
  | cor (e::es) = IFX (e, LITERAL (embedBool true), cor es)
  | cor [] = raise InternalError "parsing ||"
```

And `assert` desugars into another `IFX`:

S535d. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S535c S536a▷

```
assert: exp -> exp
```

```
fun assert e =
  IFX (e, BEGIN [], ERRORX [LITERAL (SYM "assertion-failure")])
```

The expression parser is built recursively in the usual way, using `exptable` and `exp`. Function `supriseReserved` tries to spot a common mistake: using a type or module keyword where an expression is expected.

S536a. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S535d S536b▷

```
exptable : exp parser -> exp parser
exp      : exp parser
```

```
fun exptable exp =
  let fun surpriseReserved words =
        let fun die w = ERROR ("while trying to parse an expression, I see " ^
                               "reserved word " ^ w ^
                               "... did you misspell a statement keyword earlier?")
        in  die <$>! sat (fn w => member w words) (left *> reserved)
        end
      val bindings = bindingsOf "[x e]" name exp
      val tbindings = bindingsOf "[x : ty]" formal exp
      val dbs       = distinctBsIn bindings
      val dtbs      = distinctTBsIn tbindings

      val choice   = bracket ("[pattern exp]", pair <$> pattern <*> exp)
      val body = smartBegin <$> many1 exp
      val nothing = pure (BEGIN [])

     fun lambda (xs : (name * tyex) list located) exp =
       nodupsty ("formal parameter", "lambda") xs >>=+ (fn xs => LAMBDA (xs, exp))

  in usageParsers
     [ ("(if e1 e2 e3)",            curry3 IFX          <$> exp <*> exp <*> exp)
     , ("(when e1 e ...)",          curry3 IFX          <$> exp <*> body <*> nothing)
     , ("(unless e1 e ...)",        curry3 IFX          <$> exp <*> nothing <*> body)
     , ("(set x e)",                curry  SET          <$> name <*> exp)
     , ("(while e body)",           curry  WHILEX       <$> exp  <*> body)
     , ("(begin e ...)",                   BEGIN        <$> many exp)
     , ("(error e ...)",                   ERRORX       <$> many exp)
     , ("(let (bindings) body)",    curry3 LETX LET     <$> dbs "let"    <*> body)
     , ("(let* (bindings) body)",   curry3 LETX LETSTAR <$> bindings     <*> body)
     , ("(letrec (typed-bindings) body)", curry LETRECX <$> dtbs "letrec" <*> body)
     , ("(case exp (pattern exp) ...)", curry CASE <$> exp <*> many choice)
     , ("(lambda ([x : ty] ...) body)", lambda <$> @@ lformals <*>! body)
     , ("(&& e ...)",               cand <$> many1 exp)
     , ("(|| e ...)",               cor  <$> many1 exp)
     , ("(assert e)",               assert <$> exp)
     , ("(quote sx)",               LITERAL <$> quotelit)
     ]
    <|> surpriseReserved [⟨words reserved for Molecule types S533c⟩,
                          ⟨words reserved for Molecule definitions S539a⟩]
  end
```

Molecule's expression parser has two unusual features: For overloading, every `APPLY` node allocates a ref cell. And every expression is wrapped in `EXP_AT`, which gives its location.

S536b. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S536a S537a▷

```
fun exp tokens =
  let fun applyNode f args = APPLY (f, args, ref notOverloadedIndex)
      val unlocatedExp = showErrorInput
       (  atomicExp
      <|> exptable exp
      <|> leftCurly <!> "curly brackets are not supported"
      <|> left *> right <!> "empty application"
      <|> bracket("function application", applyNode <$> exp <*> many exp)
       )
  in  EXP_AT <$> @@ unlocatedExp
  end tokens
```

T.12.2 Parsers for declarations and definitions

Both `lambda` expressions and generic modules can have typed formal parameters, so to parse them, I define `formalWith`.

S537a. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S536b S537b▷

```
formalWith : string -> 'a parser -> (name * 'a) parser
formal : (name * tyex) parser
```

```
fun formalWith whatTy aTy =
  bracket ("[x : " ^ whatTy ^ "]", pair <$> name <* kw ":" <*> aTy)

val formal = formalWith "ty" tyex
```

The `exports-record-ops` form takes a type name and a list of typed fields. Those elements are desugared into a module type by function `recordOpsType`.

S537b. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S537a S537c▷

```
recordOpsType : string -> (name * tyex) list located -> modtyex
```

```
fun recordOpsType tyname (loc, formals) =
  let val t = TYNAME (PNAME (loc, tyname))
      val unitty  = TYNAME (PDOT (PNAME (loc, "Unit"), "t"))
      val conty = FUNTY (map snd formals, t)
      fun getterty (x, tau) = (loc, (x, DECVAL (FUNTY ([t], tau))))
      fun setname x = "set-" ^ x ^ "!"
      fun setterty (x, tau) =
            (loc, (setname x, DECVAL(FUNTY ([t, tau], unitty))))
      val exports =
            (loc, (tyname, DECABSTY)) :: (loc, ("make", DECVAL conty)) ::
            map getterty formals @ map setterty formals
  in  MTEXPORTSX exports
  end
```

Similarly, the definition of a record module is desugared by function `recordModule`. Every record module includes a type definition, a constructor function, and a getter and a setter component for each field.

S537c. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S537b S538b▷

```
recordModule : name located -> name -> (name * tyex) list -> baredef
```

```
fun recordModule (loc, name) tyname fields =
  let val t = TYNAME (PNAME (loc, tyname))
      val vcon = "make-" ^ name ^ "." ^ tyname
      val conpat = CONPAT (PNAME vcon, map (PVAR o fst) fields)
      fun var x = VAR (PNAME (loc, x))
      ⟨definitions of getterComponent and setterComponent S538a⟩
      val conval = (* the record constructor *)
        LAMBDA (fields, APPLY (VCONX (PNAME vcon),
                               map (var o fst) fields,
                               ref notOverloadedIndex))

      val indices = List.tabulate (length fields, id)
      val components =
        DATA (tyname, [(vcon, FUNTY (map snd fields, t))]) ::
        VAL ("make", conval) ::
        ListPair.mapEq getterComponent (fields, indices) @
        ListPair.mapEq setterComponent (fields, indices)
      val modty = recordOpsType tyname (loc, fields)
  in  MODULE (name, MSEALED (modty, map (fn d => (loc, d)) components))
  end
```

Each getter gets the *i*th field, and the setter sets the *i*th field. It's all done by using conpat to match the record value.

S538a. ⟨*definitions of* getterComponent *and* setterComponent S538a⟩≡ (S537c)

```
fun getter i =
  (LAMBDA ([("r", t)],
           CASE (var "r",
                 [(conpat, var (fst (List.nth (fields, i))))])))
fun setter i =
  (LAMBDA ([("the record", t),
            ("the value", snd (List.nth (fields, i)))],
           CASE (var "the record",
                 [(conpat, SET (fst (List.nth (fields, i)),
                                var "the value"))])))

fun getterComponent ((x, _), i) = VAL (x, getter i)
fun setterComponent ((x, _), i) = VAL ("set-" ^ x ^ "!", setter i)
```

Below I use two forms of "right map" to change a named thing. One form takes a pair and the other is curried. In both cases a function is applied to a value that is paired with a name.

S538b. ⟨*parsers and* xdef *streams for Molecule* S532d⟩+≡ (S532b) ◁S537c S538c▷

```
prightmap : ('a -> 'b) -> name * 'a  -> name * 'b
crightmap : ('a -> 'b) -> name -> 'a -> name * 'b
```

```
fun prightmap f (x, a) = (x, f a)
fun crightmap f x a = (x, f a)
```

Another handy function acts like pair but swaps left and right.

S538c. ⟨*parsers and* xdef *streams for Molecule* S532d⟩+≡ (S532b) ◁S538b S538d▷

```
fun flipPair tx c = (c, tx)
```

```
flipPair : 'a -> 'b -> 'b * 'a
```

The four forms of declaration: abstract type, manifest type, module, and value.

S538d. ⟨*parsers and* xdef *streams for Molecule* S532d⟩+≡ (S532b) ◁S538c S539b▷

```
decl : (name * decl) parser
locmodformal : (name located * modtyex located) parser
modformal    : (name * modtyex) parser
modtype      : modtyex parser
```

```
fun decl tokens =
  (  usageParsers
       [ ("(abstype t)",          pair <$> name <*> pure DECABSTY)
       , ("(type t ty)",          crightmap DECMANTY  <$> name <*> tyex)
       , ("(module [A : modty])", prightmap DECMOD <$> modformal)
       ]
 <|> prightmap DECVAL <$> formal
  )
  tokens
and locmodformal tokens =
  bracket ("[M : modty]", pair <$> @@ name <* kw ":" <*> @@ modtype) tokens
and modformal tokens =
  ((fn (x, t) => (snd x, snd t)) <$> locmodformal) tokens
and modtype tokens = (
  usageParsers
  [ ("(exports component...)", MTEXPORTSX <$> many (@@ decl))
  , ("(allof module-type...)", MTALLOFX   <$> many (@@ modtype))
  , ("(exports-record-ops t ([x : ty] ...))",
                               recordOpsType <$> name <*> @@ lformals)
  ]
  <|> MTNAMEDX <$> name
  <|> bracket ("([A : modty] ... --m-> modty)",
               curry MTARROWX <$> many locmodformal <*> kw "--m->" *> @@ modtype)
  ) tokens
```

The definition forms reserve these words:

S539a. ⟨*words reserved for Molecule definitions* S539a⟩≡ (S531a S536a)

```
":",
"val", "define", "exports", "allof", "module-type", "module", "--m->",
"generic-module", "unsealed-module", "type", "abstype", "data",
"record-module", "exports-record-ops",
"use", "check-expect", "check-assert",
"check-error", "check-type", "check-type-error",
"check-module-type",
"overload"
```

A common mistake is to misspell a value variable for a constructor or vice versa. Or just to confuse them. I update parsers for value variables and value constructors to issue length error messages. Value variables and value constructors.

S539b. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S538d S540▷

```
vcon : name parser
vvar : name parser
```

```
fun wantedVcon (loc, x) =
  synerrorAt ("expected value constructor, but got name " ^ x) loc
fun wantedVvar (loc, x) =
  synerrorAt ("expected variable name, but got value constructor " ^ x) loc

val vvar = sat isVvar name
val vcon =
  let fun isEmptyList (left, right) =
            notCurly left andalso snd left = snd right
      val boolcon  = (fn p => if p then "#t" else "#f") <$> booltok
  in  boolcon <|> sat isVcon name <|>
      "'()" <$ quote <* sat isEmptyList (pair <$> left <*> right)
  end

val (vcon, vvar) = ( vcon <|> wantedVcon <$>! @@ vvar
                   , vvar <|> wantedVvar <$>! @@ vcon
                   )
```

All your definition forms are belong to us.

S540. ⟨*parsers and* xdef *streams for Molecule* S532d⟩+≡ (S532b) ◁S539b S541a▷

```
def   : def   parser
baredef : baredef parser
define : tyex -> name -> (name * tyex) list located -> exp -> baredef error
sealedWith : (modtyex * 'a -> moddef) -> name * modtyex -> 'a -> name * moddef
```

```
fun def tokens =
  showErrorInput (@@ baredef) tokens
and baredef tokens =
  let fun define tau f formals body =
        nodupsty ("formal parameter", "definition of function " ^ f) formals >>=+
          (fn xts => DEFINE (f, tau, (xts, body)))
      fun definestar _ = ERROR "define* is left as an exercise"
      val tyname = name
      fun valrec (x, tau) e = VALREC (x, tau, e)
      fun sealedWith f (m : name, mt : modtyex) rhs = (m, f (mt, rhs))
      val conTy = typedFormalOf vcon (kw ":") tyex
      val body : exp parser = smartBegin <$> many1 exp
  in  usageParsers
      [ ("(define type f (args) body)",
                         define <$> tyex <*> name <*> @@ lformals <*>! body)
      , ("(val x e)",    curry VAL <$> vvar <*> exp)
      , ("(val-rec [x : type] e)",  valrec <$> formal <*> exp)

      , ("(data t [vcon : ty] ...)", (curry DATA <$> tyname <*> many conTy))
      , ("(type t ty)",           curry TYPE <$> name <*> tyex)
      , ("(module-type T modty)", curry MODULETYPE <$> name <*> modtype)
      , ("(module M path) or (module [M : T] path/defs)",
            MODULE <$> (  (pair <$> name <*> MPATH <$> path)
                      <|> (sealedWith MPATHSEALED <$> modformal <*> path)
                      <|> (sealedWith MSEALED <$> modformal <*> many def)
                       ))

      , ("(generic-module [M : T] defs)",
            let fun project ((_, m), (_, t)) = (m, t)
                fun gen ((loc, M), (loc', T)) defs =
                  case T
                    of MTARROWX (formals, result) =>
                         OK (GMODULE (M, map project formals,
                                      MSEALED (snd result, defs)))
                     | _ =>
                         ERROR ("at " ^ srclocString loc' ^ ", generic " ^
                                "module " ^ M ^ " does not have an arrow type")
            in   gen <$> locmodformal <*>! many def
            end)
      , ("(unsealed-module M defs)",
            MODULE <$> (crightmap MUNSEALED <$> name <*> many def))
      , ("(record-module M t ([x : ty] ...))",
            recordModule <$> @@ name <*> name <*> lformals)
      , ("(overload qname ...)", OVERLOAD <$> many path)
      ]
     <|> QNAME <$> path
     <|> EXP <$> exp : baredef parser
  end tokens
```

Parsers for unit tests.

S541a. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S540 S541b▷

```
val testtable = usageParsers
  [ ("(check-expect e1 e2)",    curry CHECK_EXPECT     <$> exp <*> exp)
  , ("(check-assert e)",              CHECK_ASSERT     <$> exp)
  , ("(check-error e)",               CHECK_ERROR      <$> exp)
  , ("(check-type e tau)",      curry CHECK_TYPE       <$> exp <*> tyex)
  , ("(check-type-error e)",          CHECK_TYPE_ERROR <$> def)
  , ("(check-module-type M T)", curry CHECK_MTYPE      <$> path <*> modtype)
  ]
```

And the extended definitions.

S541b. ⟨*parsers and* `xdef` *streams for Molecule* S532d⟩+≡ (S532b) ◁S541a

`xdef : xdef parser`

```
fun filenameOfDotted (x, xs) =
  String.concatWith "." (x :: xs)
val xdeftable = usageParsers
  [ ("(use filename)", (USE o filenameOfDotted) <$> dotted)
  ]

val xdef =  TEST <$> testtable
        <|>          xdeftable
        <|> DEF <$>  def
        <|> badRight "unexpected right bracket"
        <?> "definition"

val xdefstream =
  interactiveParsedStream (mclToken, xdef)
```

T.13 Unit testing

Testing `check-expect` uses the overloaded definition of `=`. The index of `=` overloaded at `tau` in environment `env`, if any, is computed by function `comparisonIndex`.

S541c. ⟨*definition of* `testIsGood` *for Molecule* S541c⟩≡ (S518b) S542a▷

`comparisonIndex : binding env -> ty -> int error`

```
fun comparisonIndex env tau =
  let val wanted = FUNTY ([tau, tau], booltype)
      fun badType compty =
        (ERROR o String.concat) ["on type ", typeString tau,
                                 " operation = has type ", typeString compty]
      val index  =
        case find ("=", env)
          of ENVOVLN taus =>
               (case resolveOverloaded ("=", tau, taus)
                  of OK (compty, i) =>
                       if eqType (compty, wanted) then OK i
                       else badType compty
                   | ERROR msg => ERROR msg)
           | _ => ERROR "operator = is not overloaded, so I can't check-expect"
  in  index
  end
```

As we expect at this stage, `testIsGood` brings together shared code from other bridge languages with code written just for Molecule.

S542a. ⟨*definition of* `testIsGood` *for Molecule* S541c⟩+≡ (S518b) ◁S541c

```
                testIsGood : unit_test * (binding env * value ref env) -> bool
  fun testIsGood (test, (E, rho)) =
    let fun ty e = typeof (e, E)
                   handle NotFound x =>
                     raise TypeError ("name " ^ x ^ " is not defined")
        ⟨shared check{Expect,Assert,Error,Type{Checks, which call ty S396c⟩
        ⟨definition of checks for Molecule S542b⟩
        ⟨definition of deftystring for Molecule S543c⟩

        fun outcome e =
          withHandlers (fn () => OK (eval (e, rho))) () (ERROR o stripAtLoc)

        ⟨definition of asSyntacticValue for Molecule S543b⟩
        ⟨shared whatWasExpected S223c⟩
        ⟨shared checkExpectPassesWith, which calls outcome S224a⟩
        ⟨shared checkAssertPasses and checkErrorPasses, which call outcome S224b⟩
        ⟨definition of checkExpectPasses for Molecule S543a⟩
        ⟨definition of checkMtypePasses for Molecule S544a⟩
        ⟨shared checkTypePasses and checkTypeErrorPasses, which call ty S396a⟩

        fun passes (CHECK_EXPECT (c, e)) = checkExpectPasses (c, e)
          | passes (CHECK_ASSERT c)      = checkAssertPasses c
          | passes (CHECK_ERROR c)       = checkErrorPasses  c
          | passes (CHECK_TYPE (c, t))   = checkTypePasses   (c, elabty (t, E))
          | passes (CHECK_TYPE_ERROR (loc, c))  = atLoc loc checkTypeErrorPasses c
          | passes (CHECK_MTYPE c)       = checkMtypePasses c
    in  checks test andalso passes test
    end
```

When typechecking unit tests, Molecule has a couple of special cases:

- A `check-expect` checks only if = is overloaded at the type we need.
- A `check-type` or `check-module-types` checks only if the type (or module type) given is well formed.

S542b. ⟨*definition of* `checks` *for Molecule* S542b⟩≡ (S542a)

```
  fun checks (CHECK_EXPECT (e1, e2)) =
        checkExpectChecks (e1, e2) andalso
        (case comparisonIndex E (ty e1)
           of OK i => true
            | ERROR msg =>
                failtest ["cannot check-expect ", expString e1, ": ", msg])
    | checks (CHECK_ASSERT e)    = checkAssertChecks e
    | checks (CHECK_ERROR  e)    = checkErrorChecks  e
    | checks (CHECK_TYPE (e, t)) = (
        let val _ = elabty (t, E)
        in  true
        end handle TypeError msg =>
          failtest ["In (check-type ", expString e, " " ^ tyexString t, "), ", msg])
    | checks (CHECK_TYPE_ERROR e) = true
    | checks (CHECK_MTYPE (pathx, mt)) =
        let val path = elabpath (pathx, E)
            val _ = elabmt ((mt, path), E)
        in  true
        end handle TypeError msg =>
          failtest ["In (check-module-type ", pathexString pathx, " ",
                    mtxString mt, "), ", msg]
```

To run check-expect, I put the overloaded equality into eqfun, then apply it to the two values from checkExpectPassesWith.

S543a. ⟨*definition of* checkExpectPasses *for Molecule* S543a⟩≡ (S542a)

```
fun checkExpectPasses (c, e) =
  let val i =
        case comparisonIndex E (ty c)
          of OK i => i
           | ERROR _ => raise InternalError "overloaded = in check-expect"
      val eqfun =
        case !(find ("=", rho))
          of ARRAY vs =>
               (Array.sub (vs, i)
                handle _ => raise InternalError "overloaded subscript")
           | _ => raise InternalError "overloaded = not array"

      fun testEquals (v1, v2) =
        case eval (APPLY (LITERAL eqfun,
                          [LITERAL v1, LITERAL v2],
                          ref notOverloadedIndex),
                   rho)
          of CONVAL (PNAME "#t", []) => true
           | _ => false

  in  checkExpectPassesWith testEquals (c, e)
  end
```

To render what was expected if check-expect fails, function whatWasExpected (Appendix H) needs a definition of asSyntacticValue.

S543b. ⟨*definition of* asSyntacticValue *for Molecule* S543b⟩≡ (S542a)

asSyntacticValue : exp -> value option

```
fun asSyntacticValue (LITERAL v) = SOME v
  | asSyntacticValue (VCONX c)   = SOME (CONVAL (c, []))
  | asSyntacticValue (APPLY (e, es, _)) =
      (case (asSyntacticValue e, optionList (map asSyntacticValue es))
         of (SOME (CONVAL (c, [])), SOME vs) => SOME (CONVAL (c, map ref vs))
          | _ => NONE)
  | asSyntacticValue _ = NONE
```

The code for check-type-error is shared with other interpreters, but in order to show the outcome if a definition does *not* result in a type error, Molecule needs its own version of deftystring.

S543c. ⟨*definition of* deftystring *for Molecule* S543c⟩≡ (S542a)

deftystring : baredef -> string

```
fun deftystring d =
  let val comps = List.mapPartial asComponent (typdef (d, TOPLEVEL, E))
  in  if null comps then
        (case d of OVERLOAD _ => "an overloaded name"
                 | GMODULE _ => "a generic module"
                 | MODULETYPE _ => "a module type"
                 | _ => raise InternalError "unrecognized definition")
      else
        commaSep (map (whatcomp o snd) comps)
  end handle NotFound x => raise TypeError ("name " ^ x ^ " is not defined")
```

The check-module-type form is unique to Molecule, but its implementation is similar to that of check-type from Chapter 7.

S544a. ⟨*definition of* checkMtypePasses *for Molecule* S544a⟩≡ (S542a)

```
fun checkMtypePasses (pathx, mtx) =
  let val path = elabpath (pathx, E)
      val principal = strengthen (findModule (pathx, E), path)
      val mt = elabmt ((mtx, path), E)
  in  case implements (path, principal, mt)
        of OK () => true
         | ERROR msg => raise TypeError msg
  end handle TypeError msg =>
    failtest ["In (check-module-type ", pathexString pathx, " ",
              mtxString mtx, "), ", msg]
```

T.14 Code promised in the chapter: Histograms

Chapter 9 uses a histogram to visualize the lengths of lists in a hash table. The HISTOGRAM abstraction is defined in chunk 599b; it is implemented here. A histogram is represented by an expandable array of integers, indexed starting at zero.

S544b. ⟨*histogram.mcl* S544b⟩≡

```
(module [Histogram : (allof HISTOGRAM
                            (exports [abstype t] [print : (t -> unit)]))]
   (module A (@m ArrayList Int))
   (type t A.t)
   (type histogram t)
   (define t new ()
     (A.from 0))
   ⟨functions defined inside module Histogram S544c⟩
)
```

If inc-by is asked to increment a bucket that lies outside the bounds of the array, it enlarges the array.

S544c. ⟨*functions defined inside module* Histogram S544c⟩≡ (S544b) S544d ▷

```
(define unit inc-by ([index : int] [n : int] [a : histogram])
   (while (< index (A.lo a))
       (A.addlo a 0))
   (while (>= index (A.nexthi a))
       (A.addhi a 0))
   (A.at-put a index (+ n (A.at a index))))
```

Function inc increments by 1.

S544d. ⟨*functions defined inside module* Histogram S544c⟩+≡ (S544b) ◁ S544c S544e ▷

```
(define unit inc ([i : int] [h : histogram])
   (inc-by i 1 h))
```

If count-of is asked for the count of a bucket that lies outside the bounds of the array, it returns zero.

S544e. ⟨*functions defined inside module* Histogram S544c⟩+≡ (S544b) ◁ S544d S545a ▷

```
(define int count-of ([i : int] [a : histogram])
   (if (|| (< i (A.lo a)) (>= i (A.nexthi a)))
      0
      (A.at a i)))
```

Function `print-right` prints an index, right-justified in a three-digit field.

S545a. ⟨*functions defined inside module* `Histogram` S544c⟩+≡ (S544b) ◁S544e S545b▷

```
(define int abs ([n : int])
   (if (< n 0) (- 0 n) n))

(define unit print-right ([n : int])
   ; print in three-digit field
   (let* ([i     (abs n)]
          [bound (if (< n 0) 10 100)])
     (when (= i 0) (set i 1)) ; cheat
     (while (< i bound)
       (Int.printu 160) ; non-breaking space
       (set bound (/ bound 10)))
       (print n)))
```

Function `visualize` prints a visualization: one star for each item counted.

S545b. ⟨*functions defined inside module* `Histogram` S544c⟩+≡ (S544b) ◁S545a S545c▷

```
(define unit visualize ([a : histogram])
  (let* ([i (A.lo a)]
         [limit (A.nexthi a)])
    (while (< i limit)
       (print-right i)
       (Int.printu 32)
       (print '|)
       (let ([j 0]
             [count (A.at a i)])
          (while (< j count)
             (print '*)
             (set j (+ j 1))))
       (print Char.newline)
       (set i (+ i 1)))))
```

Function `print` can be overloaded to print just “`<histogram>`.”

S545c. ⟨*functions defined inside module* `Histogram` S544c⟩+≡ (S544b) ◁S545b

```
(define unit print ([h : histogram])
   (Sym.print '<histogram>))
```

Appendix U contents

Supporting code for μSmalltalk U

U.1 Organizing code chunks into an interpreter

The overall structure of the μSmalltalk interpreter resembles the structure of the μScheme interpreter shown in chunk S379, with the addition of chunks for bootstrapping and for stack tracing. But the organization is affected by the circularity of μSmalltalk's literals. As explained in Chapter 10 (page 697), literal Booleans, numbers, and `nil` are all instances of classes, which means they can't be used until those classes are defined. But they need to be defined before the interpreter can finish parsing the predefined classes. This need is met by placing chunk ⟨*support for bootstrapping classes/values used during parsing* S561a⟩ as early as possible: after after the definition of chunk ⟨*abstract syntax and values for μSmalltalk* S548a⟩ and the chunk that defines the associated utility functions. Then come parsing, primitives, and evaluation. The code for ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S549c⟩ comes almost at the end, just before the execution of the command line. That code reads the definitions of the rest of the predefined classes, then closes the cycles by calling the functions from ⟨*support for bootstrapping classes/values used during parsing* S561a⟩.

S547. ⟨*usm.sml* S547⟩≡

```
⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization S213a⟩

⟨abstract syntax and values for μSmalltalk S548a⟩
⟨utility functions on μSmalltalk classes, methods, and values S587c⟩

⟨support for bootstrapping classes/values used during parsing S561a⟩

⟨lexical analysis and parsing for μSmalltalk, providing filexdefs and stringsxdefs S574b⟩

⟨evaluation, testing, and the read-eval-print loop for μSmalltalk S548c⟩

⟨implementations of μSmalltalk primitives and definition of initialBasis S549c⟩
⟨function runStream for μSmalltalk, which prints stack traces S584d⟩
⟨look at command-line arguments, then run S240c⟩
```

Support for abstract syntax and values is pulled together in the same way as in the other interpreters. But in μSmalltalk, both `valueString` and `expString` use the `className` utility function, which I define here.

```
S548a. ⟨abstract syntax and values for μSmalltalk S548a⟩≡                    (S547)
  ⟨support for μSmalltalk stack frames S590a⟩
  ⟨definitions of exp, rep, and class for μSmalltalk 686a⟩
  ⟨definitions of value and method for μSmalltalk 685⟩
  ⟨definition of def for μSmalltalk 687b⟩
  ⟨definition of unit_test for μSmalltalk S548b⟩
  ⟨definition of xdef (shared) S214b⟩

  fun className (CLASS {name, ...}) = name

  ⟨definition of valueString for μSmalltalk S574a⟩
  ⟨definition of expString for μSmalltalk S573b⟩
```

Unit tests are those of μScheme, plus the `check-print` form, which is unique to μSmalltalk.

```
S548b. ⟨definition of unit_test for μSmalltalk S548b⟩≡                     (S548a)
  ⟨definition of unit_test for untyped languages (shared) S214a⟩
                 | CHECK_PRINT of exp * string
```

Evaluation involves more parts than in interpreters for other bridge languages—primarily because of the way primitives are used in the evaluator.

```
S548c. ⟨evaluation, testing, and the read-eval-print loop for μSmalltalk S548c⟩≡    (S547)
  ⟨shared definition of withHandlers S239a⟩
  ⟨definition of nullsrc, for use in ML code that sends messages S549b⟩
  ⟨support for primitives and built-in classes S550c⟩
  ⟨definition of newClassObject and supporting functions 695a⟩
  ⟨functions for managing and printing a μSmalltalk stack trace S582a⟩
  ⟨definition of primitives S550d⟩
  ⟨helper functions for evaluation S587b⟩
  ⟨definition of the Return exception 687a⟩
  ⟨evaluation, basis, and processDef for μSmalltalk S548e⟩
  ⟨shared unit-testing utilities S225a⟩
  ⟨definition of testIsGood for μSmalltalk S571a⟩
  ⟨shared definition of processTests S226⟩
  ⟨shared read-eval-print loop S237⟩
```

The predefined classes are sufficiently complicated that they need a little organization of their own. They include both numeric and collection classes.

```
S548d. ⟨predefined μSmalltalk classes and values S548d⟩≡               ◁ 655b  S560a ▷
  ⟨numeric classes S567b⟩
  ⟨predefined μSmalltalk classes and values that use numeric literals S560c⟩
  ⟨collection classes S563⟩
```

U.2 Processing definitions and building the initial basis

In μSmalltalk, as in μScheme, every name stands for a mutable location that holds a value, so basis is simply a global environment.

```
S548e. ⟨evaluation, basis, and processDef for μSmalltalk S548e⟩≡   (S548c)  ◁ 693c  S549a ▷
  type basis = value ref env
```

Extended definitions are evaluated using the reusable code presented in Chapter 5. Like μScheme, μSmalltalk works with a single top-level environment, which maps each name to a mutable location holding a value. “Processing” a definition means evaluating it, then showing the result by sending `println` to the defined value. The default `println` method calls the object's `print` method, which you can redefine.

S549a. ⟨*evaluation,* `basis`*, and* `processDef` *for μSmalltalk* S548e⟩+≡ (S548c) ◁S548e

```
fun processDef (d, xi, interactivity) =
  let val (xi', v) = evaldef (d, xi)
      val _ = if echoes interactivity then
                ignore (eval (SEND (nullsrc, VALUE v, "println", []),
                              emptyEnv, objectClass, noFrame, xi'))
              else
                ()
  in  xi'
  end
```

The source location `nullsrc` identifies the `SEND` as something generated internally, rather than read from a file or a list of strings.

S549b. ⟨*definition of* `nullsrc`*, for use in ML code that sends messages* S549b⟩≡ (S548c)

```
val nullsrc : srcloc = ("internally generated SEND node", 1)
```

The first entries in the initial basis are the primitive classes.

S549c. ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S549c⟩≡ (S547) S549d ▷

```
val initialXi = emptyEnv

fun addClass (c, xi) = bind (className c, ref (classObject c), xi)
val initialXi =
  foldl addClass initialXi [ objectClass, nilClass, classClass, metaclassClass ]
```

The next entries are the predefined classes. To help with debugging, I define function `errmsg` to identify an error as originating in a predefined class and to use `eprintlnTrace` instead of `eprintln`, so that if an error occurs, a stack trace is printed.

S549d. ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S549c⟩+≡ (S547)

```
val predefs = ⟨predefined μSmalltalk classes and values, as strings (from chunk 655b)⟩
val initialXi =
  let val xdefs = stringsxdefs ("predefined classes", predefs)
      fun errmsg s = eprintlnTrace ("error in predefined class: " ^ s)
  in  readEvalPrintWith errmsg (xdefs, initialXi, noninteractive)
      before (if logging then print "\nops.predefined_ends ()\n" else ())
  end
```

Now that the definitions of the predefined classes have been processed, it's almost time to close the cycles involving literals, blocks, and compiled methods. But first the basis has to be extended with `VAL` bindings for `true` and `false`. Because the parser prevents user code from binding `true` and `false`, these bindings can't be created using μSmalltalk code; instead, the bindings are added to `initialXi` using ML code.

S549e. ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S549c⟩+≡ (S547)

```
fun addVal x e xi = processDef (VAL (x, e), xi, noninteractive)
                                  addVal : name -> exp -> basis -> basis

local
  fun newInstance classname = SEND (nullsrc, VAR classname, "new", [])
in
  val initialXi = addVal "true"  (newInstance "True" ) initialXi
  val initialXi = addVal "false" (newInstance "False") initialXi
end
```

Now the cycles can be closed. All the necessary classes should be defined, so if any cycle fails to close, the interpreter halts with a fatal error.

S550a. ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S549c⟩+≡ (S547) ◁S549e S550b

```
val _ =
  ( saveLiteralClasses      initialXi
  ; saveTrueAndFalse        initialXi
  ; saveBlockClass          initialXi
  ; saveCompiledMethodClass initialXi
  ) handle NotFound n =>
      ( app eprint ["Fatal error: ", n, " is not predefined\n"]
      ; raise InternalError "this can't happen"
      )
  | e => ( eprintln "Error binding predefined classes into interpreter"; raise e)
```

The last step of initialization is to bind the predefined value `nil`. Like bindings for `true` and `false`, a `val` binding for `nil` can't be parsed, so the binding is added using ML code.

S550b. ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S549c⟩+≡ (S547) ◁S550a

```
val initialXi = addVal "nil" (VALUE nilValue) initialXi
val initialBasis = initialXi
val primitiveBasis = primitives
```

U.3 Primitives

The definitions of the primitives are carefully layered. The first layer includes chunks ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S552a⟩ and ⟨*built-in class* `Object` 696a⟩. This code defines `Object`, which enables me to define `UndefinedObject` and `nilValue` (the internal representation of `nil`) in the second layer, in chunk ⟨*built-in class* `UndefinedObject` *and value* `nilValue` 696b⟩. The third layer includes chunks ⟨*ML code for remaining classes' primitives* S551b⟩ and ⟨*built-in classes* `Class` *and* `Metaclass` 696d⟩. They define all the other primitives and built-in classes, some of which use `nilValue`.

S550c. ⟨*support for primitives and built-in classes* S550c⟩≡ (S548c)

⟨*utility functions for building primitives in μSmalltalk* S551a⟩
⟨*metaclass utilities* S588d⟩
⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S552a⟩
⟨*utility functions for parsing internal method definitions* S558a⟩
⟨*built-in class* `Object` 696a⟩
⟨*built-in class* `UndefinedObject` *and value* `nilValue` 696b⟩
⟨*higher-order functions for creating μSmalltalk primitives* S554h⟩
⟨*ML code for remaining classes' primitives* S551b⟩
⟨*built-in classes* `Class` *and* `Metaclass` 696d⟩
⟨*metaclasses for built-in classes* 695c⟩

A primitive is found by looking up its name in the association list `primitives`.

S550d. ⟨*definition of* `primitives` S550d⟩≡ (S548c)

```
val primitives = ⟨primitives for μSmalltalk :: S551c⟩ nil
```

U.3.1 Utilities for creating primitives

Most primitives are created directly from ML functions. As in the interpreter for *μ*Scheme (Chapter 5), I organize the code into stages. The first stage turns unary and binary functions into primitives.

S551a. ⟨*utility functions for building primitives in μSmalltalk* S551a⟩≡ (S550c)

```
unaryPrim  : (value         -> value) -> primitive
binaryPrim : (value * value -> value) -> primitive
```

```
type primitive = value list * value ref env -> value
fun arityError n args =
  raise RuntimeError ("primitive expected " ^ intString n ^
                      " arguments; got " ^ intString (length args))
fun unaryPrim  f = (fn ([a],    _) => f  a     | (vs, _) => arityError 0 vs)
fun binaryPrim f = (fn ([a, b], _) => f (a, b) | (vs, _) => arityError 1 vs)
```

U.3.2 Primitives for arithmetic with overflow

To detect overflow in arithmetic, the arithmetic primitives catch ML's predefined `Overflow` exception. Each relevant primitive tries to build a block containing the result of the arithmetic, but if `Overflow` is raised, the primitive answers the overflow block instead. The logic is implemented once and for all by function `withOverflow`. Function `internalThunk` builds a block from a list of expressions.

S551b. ⟨*ML code for remaining classes' primitives* S551b⟩≡ (S550c) ◁700 S553e ▷

```
internalThunk : exp list -> value
withOverflow : (int * int -> int) -> primitive
```

```
fun withOverflow binop ([(_, NUM n), (_, NUM m), ovflw], xi) =
      (internalThunk [VALUE (mkInteger (binop (n, m)))]
       handle Overflow => ovflw)
  | withOverflow _ ([_, _, _], _) =
      raise RuntimeError "numeric primitive with overflow expects numbers"
  | withOverflow _ _ =
      raise RuntimeError "numeric primitive with overflow expects 3 arguments"
and internalThunk exps =
      mkBlock ([], exps, emptyEnv, objectClass, noFrame)
```

S551c. ⟨*primitives for μSmalltalk* :: S551c⟩≡ (S550d) S551d ▷

```
("addWithOverflow", withOverflow op + ) ::
("subWithOverflow", withOverflow op - ) ::
("mulWithOverflow", withOverflow op * ) ::
```

U.3.3 Hashing

The hash primitive works only on symbols. It uses the hash function of Fowler, Vo, and Noll, from Appendix H (page S215).

S551d. ⟨*primitives for μSmalltalk* :: S551c⟩+≡ (S550d) ◁S551c S552b ▷

```
("hash", unaryPrim
             (fn (_, SYM s) => mkInteger (fnvHash s)
               | v => raise RuntimeError "hash primitive expects a symbol")) ::
```

U.3.4 Object primitives

Object identity

The `sameObject` primitive decides whether objects are identical by comparing their representations, using ML function `eqRep`.

S552a. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S552a⟩≡ (S550c) S552c ▷

```
eqRep : value * value -> bool
```

```
fun eqRep ((cx, x), (cy, y)) =
  classId cx = classId cy andalso
  case (x, y)
    of (ARRAY x,    ARRAY    y) => x = y
     | (NUM   x,    NUM      y) => x = y
     | (SYM   x,    SYM      y) => x = y
     | (USER  x,    USER     y) => x = y
     | (CLOSURE  x, CLOSURE  y) => false
     | (CLASSREP x, CLASSREP y) => classId x = classId y
     | _ => false
```

S552b. ⟨*primitives for μSmalltalk* `::` S551c⟩+≡ (S550d) ◁ S551d S553a ▷

```
("sameObject", binaryPrim (mkBoolean o eqRep)) ::
```

Each case is justified as follows:

- ML equality on arrays *is* object identity.
- Because numbers and symbols are immutable in both Smalltalk and ML, numbers and symbols that compare equal are indistinguishable by any μSmalltalk program.
- The `USER` representation is an environment containing mutable reference cells. ML's `ref` function is also generative, so ML equality on ref cells is object identity. Comparing the representation of two `USER` objects compares their instance-variable environments, which are equal only if they contain the same ref cells, which is possible only if they represent the same μSmalltalk object.
- Blocks, which are represented as closures, can't easily be compared, because the body of a block may contain a literal primitive function, and ML equality can't compare functions. A block is therefore not equal to anything, not even itself.
- Two classes are the same object if and only if they have the same unique identifier.

Class membership

For primitive `memberOf`, the class `c` of `self` has to be the same as the class `c'` of the argument. For `kindOf`, it just has to be a subclass.

S552c. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S552a⟩+≡ (S550c) ◁ S552a S553c ▷

```
fun memberOf ((c, _), (_, CLASSREP c')) = mkBoolean (classId c = classId c')
  | memberOf _ = raise RuntimeError "argument of isMemberOf: must be a class"

fun kindOf ((c, _), (_, CLASSREP (CLASS {class=u', ...}))) =
      let fun subclassOfClassU' (CLASS {super, class=u, ... }) =
            u = u' orelse (case super of NONE => false
                                      | SOME c => subclassOfClassU' c)
      in  mkBoolean (subclassOfClassU' c)
      end
  | kindOf _ = raise RuntimeError "argument of isKindOf: must be a class"
```

S553a. ⟨*primitives for µSmalltalk* :: S551c⟩+≡ (S550d) ◁S552b S553b▷

```
("isKindOf",   binaryPrim kindOf) ::
("isMemberOf", binaryPrim memberOf) ::
```

Class inquiry

The `class` primitive is implemented by function `classPrimitive`, which is defined in Chapter 10 (page 699).

S553b. ⟨*primitives for µSmalltalk* :: S551c⟩+≡ (S550d) ◁S553a S553d▷

```
("class", classPrimitive) ::
```

Error messages

The `error:` primitive raises `RuntimeError`.

S553c. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S552a⟩+≡ (S550c) ◁S552c S559a▷

```
fun error (_, (_, SYM s)) = raise RuntimeError s
  | error (_, (c, _    )) =
      raise RuntimeError ("error: got class " ^ className c ^ "; expected Symbol")
```

S553d. ⟨*primitives for µSmalltalk* :: S551c⟩+≡ (S550d) ◁S553b S553g▷

```
("error", binaryPrim error) ::
```

U.3.5 Integer primitives

Integers print using the `intString` function defined in Appendix H.

S553e. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S551b S553f▷

```
fun printInt (self as (_, NUM n)) = ( xprint (intString n); self )
  | printInt _ = raise RuntimeError ("printInt primitive on non-SmallInteger")
```

Integers also support UTF-8 printing.

S553f. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S553e S553h▷

```
fun printu (self as (_, NUM n)) = ( printUTF8 n; self )
  | printu _ = raise RuntimeError ("printu primitives on non-SmallInteger")
```

S553g. ⟨*primitives for µSmalltalk* :: S551c⟩+≡ (S550d) ◁S553d S554a▷

```
("printSmallInteger", unaryPrim printInt) ::
("printu",            unaryPrim printu)   ::
```

A binary integer operation created with `arith` expects as arguments two integers `m` and `n`; it applies an `operator` to them and uses a creator function `mk` to convert the result to a value. I use `binaryInt` to build arithmetic and comparison.

S553h. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S553f S554b▷

```
binaryInt  : ('a -> value) -> (int * int -> 'a)   -> value * value -> value
arithop    :                  (int * int -> int)  -> primitive
intcompare :                  (int * int -> bool) -> primitive
```

```
fun binaryInt mk operator ((_, NUM n), (_, NUM m)) = mk (operator (n, m))
  | binaryInt _ _         ((_, NUM n), (c, _)) =
      raise RuntimeError ("numeric primitive expected numeric argument, got <"
                          ^ className c ^ ">")
  | binaryInt _ _         ((c, _), _) =
      raise RuntimeError ("numeric primitive method defined on <" ^ className c ^ ">")
fun arithop    operator = binaryPrim (binaryInt mkInteger operator)
fun intcompare operator = binaryPrim (binaryInt mkBoolean operator)
```

These functions are used to define the primitive operations on small integers.

S554a. ⟨*primitives for μSmalltalk* :: S551c⟩+≡ (S550d) ◁S553g S554c▷

```
("+",   arithop op +  )  ::
("-",   arithop op -  )  ::
("*",   arithop op *  )  ::
("div", arithop op div)  ::
("<",   intcompare op <) ::
(">",   intcompare op >) ::
```

To implement class method `new:` on `SmallInteger` requires a primitive. This primitive must receive the class, plus an argument that is represented by the integer being created.

S554b. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S553h S554d▷

```
fun newInteger ((_, CLASSREP c), (_, NUM n)) = (c, NUM n)
  | newInteger _ = raise RuntimeError ("made new integer with non-int or non-class")
```

S554c. ⟨*primitives for μSmalltalk* :: S551c⟩+≡ (S550d) ◁S554a S554f▷

```
("newSmallInteger", binaryPrim newInteger) ::
```

The primitives above are used to define class `SmallInteger` (chunk S567b).

U.3.6 Symbol primitives

A symbol prints as its name, with no leading `'`.

S554d. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S554b S554e▷

```
fun printSymbol (self as (_, SYM s)) = (xprint s; self)
  | printSymbol _ = raise RuntimeError "cannot print when object inherits from Symbol"
```

Like `new:` on `SmallInteger`, method `new:` on class `Symbol` also needs its own primitive.

S554e. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S554d S554g▷

```
fun newSymbol ((_, CLASSREP c), (_, SYM s)) = (c, SYM s)
  | newSymbol _ = raise RuntimeError ("made new symbol with non-symbol or non-class")
```

S554f. ⟨*primitives for μSmalltalk* :: S551c⟩+≡ (S550d) ◁S554c S555b▷

```
("printSymbol", unaryPrim  printSymbol) ::
("newSymbol",   binaryPrim newSymbol  ) ::
```

U.3.7 Array primitives

The primitive operations on arrays are creation, subscript, update, and size.

In a new array, every element is initialized to `nil`.

S554g. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S554e S554i▷

```
fun newArray ((_, CLASSREP c), (_, NUM n)) = (c, ARRAY (Array.array (n, nilValue)))
  | newArray _ = raise RuntimeError "Array new sent to non-class or got non-integer"
```

Each array primitive expects `self` to be an array. This expectation is checked by higher-order function `arrayPrimitive`.

S554h. ⟨*higher-order functions for creating μSmalltalk primitives* S554h⟩≡ (S550c) S555e▷

```
arrayPrimitive : (value array * value list -> value) -> primitive
```

```
fun arrayPrimitive f ((c, ARRAY a) :: vs, _) = f (a, vs)
  | arrayPrimitive f _ = raise RuntimeError "Array primitive used on non-array"
```

Each array primitive is built from a function that takes a `value array` as its first argument. Starting with `arraySize`.

S554i. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S554g S555a▷

```
fun arraySize (a, []) = mkInteger (Array.length a)
  | arraySize (a, vs) = arityError 0 vs
```

The array primitives for `at:` and `at:put:` use Standard ML's `Array` module.

S555a. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S554i S556a▷

```
fun arrayAt (a, [(_, NUM n)]) = Array.sub (a, n)
  | arrayAt (_, [_])  = raise RuntimeError "Non-integer used as array subscript"
  | arrayAt (_, vs)   = arityError 1 vs

fun arrayUpdate (a, [(_, NUM n), x]) = (Array.update (a, n, x); nilValue)
  | arrayUpdate (_, [_, _]) = raise RuntimeError "Non-integer used as array subscript"
  | arrayUpdate (_, vs)     = arityError 2 vs
```

The functions above are used to define the array primitives.

S555b. ⟨*primitives for μSmalltalk* `::` S551c⟩+≡ (S550d) ◁S554f S555c▷

```
("arrayNew",    binaryPrim     newArray)   ::
("arraySize",   arrayPrimitive arraySize)  ::
("arrayAt",     arrayPrimitive arrayAt)    ::
("arrayUpdate", arrayPrimitive arrayUpdate) ::
```

In chunk S565e, these primitive methods are used to define class `Array`.

U.3.8 Block primitives

The only primitive a block needs is `value`, which is defined in Chapter 10 (chunk 691b).

S555c. ⟨*primitives for μSmalltalk* `::` S551c⟩+≡ (S550d) ◁S555b S555d▷

```
("value", valuePrim) ::
```

U.3.9 Class primitives

The following primitives are used in class objects. Their implementations appear below.

S555d. ⟨*primitives for μSmalltalk* `::` S551c⟩+≡ (S550d) ◁S555c S558d▷

```
("protocol",      classPrim (protocols true)) ::
("localProtocol", classPrim (protocols false)) ::
("newUserObject", newPrimitive) ::
("superclass",    classPrim superclassObject) ::
("className",     classPrim (fn (_, c) => mkSymbol (className c))) ::
("getMethod",     binaryPrim getMethod) ::
("setMethod",     setMethod o fst) ::
("removeMethod",  binaryPrim removeMethod) ::
("methodNames",   classPrim methodNames) ::
```

A primitive that expects a class argument can be made using ML function `classPrim`, which builds a primitive from an ML function that takes both the metaclass and class object of the argument.

S555e. ⟨*higher-order functions for creating μSmalltalk primitives* S554h⟩+≡ (S550c) ◁S554h

```
fun classPrim f =          classPrim : (class * class -> value) -> primitive
  unaryPrim (fn (meta, CLASSREP c) => f (meta, c)
              | _ => raise RuntimeError "class primitive sent to non-class")
```

Showing protocols

The `showProtocol` function helps implement the `protocol` and `localProtocol` primitives, which are used to implement the methods of the same names on class `Class`. The implementation of `showProtocol` is not very interesting. Function insert helps implement an insertion sort, which is used to present methods in alphabetical order.

S556a. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁ S555a S556b ▷

```
local
  fun showProtocol doSuper kind c =
    let fun member x l = List.exists (fn x' : string => x' = x) l
        fun insert (x, []) = [x]
          | insert (x, (h::t)) =
              case compare x h
                of LESS    => x :: h :: t
                 | EQUAL   => x :: t (* replace *)
                 | GREATER => h :: insert (x, t)
        and compare (name, _) (name', _) = String.compare (name, name')
        fun methods (CLASS { super, methods = ref ms, name, ... }) =
              if     not doSuper
              orelse (kind = "class-method" andalso name = "Class")
              then
                foldl insert [] ms
              else
                foldl insert (case super of NONE => [] | SOME c => methods c) ms
        fun show (name, { formals, ... } : method) =
              app xprint ["(", kind, " ", name,
                          " (", spaceSep formals, ") ...)\n"]
    in  app show (methods c)
    end
in
  fun protocols all (meta, c) =
    ( showProtocol all "class-method" meta
    ; showProtocol all "method" c
    ; (meta, CLASSREP c)
    )
end
```

Reflection: Manipulating a class's methods

The remaining functions implement the primitives that query, add, and remove methods from a class object.

S556b. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁ S556a S556c ▷

```
fun methodNames (_, CLASS { methods, ... }) = mkArray (map (mkSymbol o fst) (!methods))
```

S556c. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁ S556b S557a ▷

```
fun getMethod ((_, CLASSREP (c as CLASS { methods, name, ... })), (_, SYM s)) =
      (mkCompiledMethod (find (s, !methods))
       handle NotFound _ =>
         raise RuntimeError ("class " ^ className c ^ " has no method " ^ s))
      before (if logging then logGetMethod name s else ())
  | getMethod ((_, CLASSREP _), _) =
      raise RuntimeError "getMethod primitive given non-name"
  | getMethod _ =
      raise RuntimeError "getMethod primitive given non-class"
```

S557a. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S556c S557b▷

```
fun removeMethod ((_, CLASSREP (c as CLASS { methods, ... })), (_, SYM s)) =
      (methods := List.filter (fn (m, _) => m <> s) (!methods); nilValue)
  | removeMethod ((_, CLASSREP _), _) =
      raise RuntimeError "removeMethod primitive given non-name"
  | removeMethod _ =
      raise RuntimeError "removeMethod primitive given non-class"
```

In setMethod, the ellipsis (...) in the binding for formals, locals, and body is a nifty way of binding just some of the fields in a Standard ML record. Local function given handles the error message if the setMethod primitive is given the wrong number or classes of arguments.

S557b. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S557a S559c▷

```
local
  fun given what = raise RuntimeError ("setMethod primitive given " ^ what)
in
  fun setMethod [(_, CLASSREP c), (_, SYM s), (_, METHODV m)] =
        let val CLASS { methods, super, name = cname, ... } = c
            val superclass = case super of SOME s => s | NONE => c (* bogus *)
            val { formals = xs, locals = ys, body = e, ... } = m
            val m' = { name = s, formals = xs, locals = ys, body = e
                     , superclass = superclass }
            val _ = if badColon s then
                        raise RuntimeError ("compiled method cannot have " ^
                                            "symbolic name " ^ s ^
                                            ", which has a colon")
                    else ()
            val _ = if arity s = length xs then ()
                    else raise RuntimeError ("compiled method with " ^
                                             countString xs "argument" ^
                                             " cannot have name `" ^ s ^ "`")
            val _ = if logging then logSetMethod cname s else ()
        in  (methods := bind (s, m', !methods); nilValue)
        end
    | setMethod [(_, CLASSREP _), (_, SYM s), m] =
                            given ("non-method " ^ valueString m)
    | setMethod [(_, CLASSREP _), s, _] =
                            given ("non-symbol " ^ valueString s)
    | setMethod [c, _, _] = given ("non-class " ^ valueString c)
    | setMethod _ =         given "wrong number of arguments"
end
```

arity	S575b
badColon	S576a
bind	305d
CLASS	686c
className	S588a
CLASSREP	686a
countString	S214d
find	305b
fst	S249b
type method	686d
METHODV	686a
mkArray	698a
mkCompiledMethod	S570c
mkSymbol	698a
nilValue	696c
NotFound	305b
RuntimeError	S213b
spaceSep	S214e
SYM	686a
valueString	S574a
xprint	S215d

U.4 Remaining implementations of primitive classes

The remaining methods of the primitive classes are written in μSmalltalk. Just like methods in predefined or user-defined classes, the code for these methods needs to be parsed—but the μSmalltalk code lives in string literals in the interpreter's source, not in an external file. So this part of the interpreter needs a parser for methods represented as strings.

Parsing begins with `internalParse`, an auxiliary function that applies any parser to a list of strings.

S558a. ⟨*utility functions for parsing internal method definitions* S558a⟩≡ (S550c) S558b ▷

```
                                          internalParse : 'a parser -> string list -> 'a
  fun internalParse parser ss =
    let val synopsis = case ss of [s] => s
                                | ["(begin ", s, ")"] => s
                                | s :: ss => s ^ "..."
                                | [] => ""
        val name = "internal syntax"
        val input = interactiveParsedStream (smalltalkToken, parser)
                                            (name, streamOfList ss, noPrompts)
        exception BadUserMethodInInterpreter of string (* can't be caught *)
    in  case streamGet input
          of SOME (e, _) => e
           | NONE => (app eprintln ("Failure to parse:" :: ss)
                     ; raise BadUserMethodInInterpreter (concat ss))
    end
```

The strings are converted to methods by function `internalMethods`.

S558b. ⟨*utility functions for parsing internal method definitions* S558a⟩+≡ (S550c) ◁ S558a

```
                                          internalMethods : string list -> method list
  val bogusSuperclass =
    CLASS { name = "bogus superclass", super = NONE
          , ivars = [], methods = ref [ ], class = ref PENDING
          }
  val internalMethodDefns = methodDefns (bogusSuperclass, bogusSuperclass)
  fun internalMethods strings =
    case (internalMethodDefns o internalParse parseMethods) strings
      of ([], imethods) => imethods
       | (_ :: _, _) => raise InternalError "primitive class has class methods"
```

In Chapter 10, methods of classes `Object`, `Class`, `UndefinedObject`, and `Metaclass` are defined by applying `internalMethods` to μSmalltalk code stored in strings. Some of those methods are defined in Chapter 10. The remaining methods are defined in the rest of this section.

U.4.1 Class `Object`

The methods of class `Object` that are *not* defined in Chapter 10 are defined here.

S558c. ⟨*methods of class* `Object` S558c⟩≡ ◁ 655a

```
  (method print   ()          ('< print) (((self class) name) print) ('> print) self)
  (method println ()          (self print) (newline print) self)
  (method class   ()          (primitive class self))
  (method isKindOf:  (aClass) (primitive isKindOf self aClass))
  (method isMemberOf:(aClass) (primitive isMemberOf self aClass))
  (method error:     (msg)    (primitive error self msg))
  (method subclassResponsibility () (primitive subclassResponsibility self))
  (method leftAsExercise () (primitive leftAsExercise self))
```

Methods `subclassResponsibility` and `leftAsExercise` are actually implemented by primitives.

S558d. ⟨*primitives for μSmalltalk* :: S551c⟩+≡ (S550d) ◁ S555d

```
  ("subclassResponsibility",
      errorPrim "subclass failed to implement a method it was responsible for") ::
  ("leftAsExercise",
      errorPrim "method was meant to be implemented as an exercise") ::
```

S559a. ⟨*ML functions for* Object*'s and* UndefinedObject*'s primitives* S552a⟩+≡ (S550c) ◁S553c S588c▷

```
fun errorPrim msg = fn _ => raise RuntimeError msg
```

U.4.2 Class Class

The methods of class Class that are *not* defined in Chapter 10 are defined here. The rest are defined here.

S559b. ⟨*methods of class* Class S559b⟩≡ ◁697a

```
(method superclass         () (primitive superclass self))
(method name               () (primitive className self))
(method printProtocol      () (primitive protocol self))
(method printLocalProtocol () (primitive localProtocol self))
(method methodNames        () (primitive methodNames self))
(method compiledMethodAt: (aSymbol) (primitive getMethod self aSymbol))
(method addSelector:withMethod: (aSymbol aMethod)
    (primitive setMethod self aSymbol aMethod)
    self)
(method removeSelector: (aSymbol)
   (primitive removeMethod self aSymbol)
   self)
```

For the superclass primitive, a class's superclass, if it has one, is taken right out of its representation.

S559c. ⟨*ML code for remaining classes' primitives* S551b⟩+≡ (S550c) ◁S557b

```
fun superclassObject (_, CLASS { super = NONE, ... })   = nilValue
  | superclassObject (_, CLASS { super = SOME c, ... }) = classObject c
```

U.4.3 Class UndefinedObject

The nil object gets a special print method. The other methods that are defined on class UndefinedObject are defined in Chapter 10.

S559d. ⟨*methods of class* UndefinedObject S559d⟩≡ ◁654

```
(method print () ('nil print) self)
```

U.5 Implementations of predefined classes

While many of the predefined classes use primitives, none of their methods are implemented in ML—μSmalltalk code is sufficient.

U.5.1 *Implementation of blocks*

A block is an abstraction of a function, and its representation is primitive. The `value` method is also primitive, but the `while`, `whileTrue:`, and `whileFalse:` methods are easily defined in ordinary μSmalltalk.

S560a. ⟨*predefined μSmalltalk classes and values* S548d⟩+≡ ◁S548d S561c▷

```
(class Block
    [subclass-of Object] ; internal representation
    (class-method new () {})
    (method value              ()         (primitive value self))
    (method value:             (a1)       (primitive value self a1))
    (method value:value:       (a1 a2)    (primitive value self a1 a2))
    (method value:value:value: (a1 a2 a3) (primitive value self a1 a2 a3))
    (method value:value:value:value: (a1 a2 a3 a4)
        (primitive value self a1 a2 a3 a4))
    (method whileTrue: (body)
        ((self value) ifTrue:ifFalse:
            {(body value)
             (self whileTrue: body)}
            {nil}))
    (method whileFalse: (body)
         ((self value) ifTrue:ifFalse:
             {nil}
             {(body value)
              (self whileFalse: body)}))
    ⟨tracing methods on class Block S560b⟩
)
```

Message tracing is turned on by `messageTraceFor:` or `messageTrace`.

S560b. ⟨*tracing methods on class* `Block` S560b⟩≡ (S560a)

```
(method messageTraceFor: (n) [locals answer]
    (set &trace n)
    (set answer (self value))
    (set &trace 0)
    answer)
(method messageTrace () (self messageTraceFor: -1))
```

The `&trace` referred to is a global variable, normally 0.

S560c. ⟨*predefined μSmalltalk classes and values that use numeric literals* S560c⟩≡ (S548d) S562a▷

```
(val &trace 0)
```

Bootstrapping blocks

The `Block` class doesn't actually have to be bootstrapped, but by defining `Block` and `Boolean` together (and bootstrapping them both), I clarify their relationship. That clarity is especially important to the implementations of the `whileTrue:` and `whileFalse:` methods.

S561a. ⟨*support for bootstrapping classes/values used during parsing* S561a⟩≡ (S547) ◁699a S570c▷

```
  local                 mkBlock : name list * exp list * value ref env * class * frame -> value
    val blockClass = ref NONE : class option ref
  in
    fun mkBlock c = (valOf (!blockClass), CLOSURE c)
      handle Option =>
          raise InternalError
              "Bad blockClass; evaluated block expression in predefined classes?"
    fun saveBlockClass xi =
      blockClass := SOME (findClass ("Block", xi))
  end
```

U.5.2 Implementation of `Boolean`

Class `True` is implemented in Chapter 10 (page 655), and class `False` is an exercise. Their common superclass, `Boolean`, is just boring.

S561b. ⟨*definition of class* `Boolean` S561b⟩≡

```
  (class Boolean
      [subclass-of Object]
      (method ifTrue:ifFalse: (trueBlock falseBlock)
                                      (self subclassResponsibility))
      (method ifFalse:ifTrue: (falseBlock trueBlock)
                                      (self subclassResponsibility))
      (method ifTrue:  (trueBlock)    (self subclassResponsibility))
      (method ifFalse: (falseBlock)   (self subclassResponsibility))

      (method not ()                  (self subclassResponsibility))
      (method eqv: (aBoolean)         (self subclassResponsibility))
      (method xor: (aBoolean)         (self subclassResponsibility))
      (method & (aBoolean)            (self subclassResponsibility))
      (method | (aBoolean)            (self subclassResponsibility))

      (method and: (alternativeBlock) (self subclassResponsibility))
      (method or:  (alternativeBlock) (self subclassResponsibility))
  )
```

U.5.3 Implementation of `Symbol`

Symbols really exist only to be printed and hashed.

S561c. ⟨*predefined µSmalltalk classes and values* S548d⟩+≡ ◁S560a S570b▷

```
  (class Symbol
      [subclass-of Object] ; internal representation
      (class-method new  () (self error: 'can't-send-new-to-Symbol))
      (class-method new: (aSymbol) (primitive newSymbol self aSymbol))
      (method       print  () (primitive printSymbol self))
      (method       hash   () (primitive hash self))
  )
```

U.5.4 Implementation of `Char`*: Unicode characters*

As in the other bridge languages, a Unicode character prints using the UTF-8 encoding. The `Char` class defines a representation, initialization methods, and a `print` method. It must also redefine `=`, because two objects that represent the same Unicode character should be considered equal, even if they are not the same object. The representation invariant is that `code-point` is an integer between 0 and hexadecimal `1fffff`.

S562a. ⟨*predefined μSmalltalk classes and values that use numeric literals* S560c⟩+≡ (S548d) ◁ S560c S562b ▷

```
(class Char
   [subclass-of Object]
   [ivars code-point]
   (class-method new: (n) ((self new) init: n))
   (method init:      (n) (set code-point n) self) ;; private
   (method print      ()  (primitive printu code-point))
   (method =          (c) (code-point = (c code-point)))
   (method code-point ()  code-point) ;; private
)
```

The predefined characters are defined using their code points, which coincide with 7-bit ASCII codes.

S562b. ⟨*predefined μSmalltalk classes and values that use numeric literals* S560c⟩+≡ (S548d) ◁ S562a

```
(val newline      (Char new: 10))   (val left-round   (Char new:  40))
(val space        (Char new: 32))   (val right-round  (Char new:  41))
(val semicolon    (Char new: 59))   (val left-curly   (Char new: 123))
(val quotemark    (Char new: 39))   (val right-curly  (Char new: 125))
                                    (val left-square  (Char new:  91))
                                    (val right-square (Char new:  93))
```

U.5.5 Implementation of `Set`

`Set` is a concrete class: it has instances. And an instance of `Set` is an abstraction, so all the technology from Chapter 9 comes into play: to implement `Set`, I need to know what the abstraction is, what the representation is, what the abstraction function is, what the representation invariant is, and what operations need to be implemented.

The abstraction is a set of objects. Like most other Smalltalk collections, a `Set` is mutable; for example, sending `add:` to a set changes the set. The representation is a list containing the members of the set; that list is stored in a single instance variable, `members`. The list is represented by a `List` object; this structure makes `Set` a *client* of `List`, not a subclass or superclass. The abstraction function takes the list of members and returns the set containing exactly those members. The representation invariant is that `members` contains no repeated elements.

The abstraction, representation, abstraction function, and invariant are as they would be in a language with abstract data types. But the operations that need to be implemented are different. It is true that a `Set` object needs to implement everything in its interface, which means the entire `Collection` protocol. But it doesn't do all the work itself: almost all of the protocol is implemented in class

Collection, and Set inherits those implementations. The only methods that *must* be implemented in Set are the "subclass responsibility" methods do:, add:, remove:ifAbsent:, =, and species, plus the private method printName.

S563. ⟨*collection classes* S563⟩≡ (S548d) ◁ 674a S564a ▷

```
(class Set
    [subclass-of Collection]
    [ivars members]  ; list of elements [invariant: no repeats]
    (class-method new () ((super new) initSet))
    (method initSet   () (set members (List new)) self) ; private
    (method do: (aBlock) (members do: aBlock))
    (method add: (item)
        ((members includes: item) ifFalse: {(members add: item)})
        self)
    (method remove:ifAbsent: (item exnBlock)
        (members remove:ifAbsent: item exnBlock)
        self)
    (method = (s)
      (((self size) = (s size)) ifFalse:ifTrue:
         { (return false) }
         { (self do: [block (x) ((s includes: x) ifFalse: {(return false)})])
           (return true)
         }))
)
```

To better understand how a concrete Collection class is implemented, let's look at each method.

- The class method new initializes the representation (to the empty list) by means of private instance method initSet.
- Two of the five methods required of a subclass, do: and remove:ifAbsent:, are implemented by sending the same message to members. We say these messages are *delegated* to class List.
- The required add: method cannot be delegated to List, because a set must avoid duplicates in members. To avoid duplicates, the add: method first sends the includes: message to members; item is added members only if includes: answers false. It would also work if add: sent the includes: message to self, but because List might have an includes: method that is more efficient than the default version that self inherits from Collection, Set sends includes: to members instead.
- The required = method cannot be delegated, because two sets can be equivalent even if their representations are not. Equivalence is independent of order; two sets are equivalent if they contain the same elements. It is sufficient to know that both sets are of the same size, and one contains all the elements found in the other. The implementation uses (return false) to terminate the method the moment a non-matching element is found.

In addition to the methods shown in the class definition, class Set inherits size, isEmpty, includes:, print, and other methods from Collection.

U.5.6 Implementation of `Association`

Method `associationsDo:` visits all the key-value pairs in a keyed collection. A key-value pair is represented by an object of class `Association`.

S564a. ⟨*collection classes* S563⟩+≡ (S548d) ◁S563 S564b▷

```
(class Association
   [subclass-of Object]
   [ivars key value]
   (class-method withKey:value: (x y) ((self new) setKey:value: x y))
   (method setKey:value: (x y) (set key x) (set value y) self) ; private
   (method key       ()  key)
   (method value     ()  value)
   (method setKey:   (x) (set key   x))
   (method setValue: (y) (set value y))
   (method =         (a) ((key = (a key)) & (value = (a value))))
)
```

`Associations` are mutable.

U.5.7 Implementation of `Dictionary`

A `Dictionary` is the simplest and least specialized of the keyed collections. If all μSmalltalk objects could be hashed, I would want to represent a `Dictionary` as a hash table. Because not every μSmalltalk object can be hashed, I use a list of `Associations` instead. The abstraction is a finite map, which is to say, a function with a finite domain. The representation is a list of `Associations` stored in instance variable `table`. The representation invariant is that in `table`, no single `key` appears in more than one `Association`. The abstraction function takes the representation to the function that is undefined on all `keys` not in `table` and that maps each `key` in `table` to the corresponding `value`.

S564b. ⟨*collection classes* S563⟩+≡ (S548d) ◁S564a S565e▷

```
(class Dictionary
    [subclass-of KeyedCollection]
    [ivars table] ; list of Associations
    (class-method new ()      ((super new) initDictionary))
    (method initDictionary () (set table (List new)) self) ; private
    ⟨other methods of class Dictionary S564c⟩
)
```

The operations that `Dictionary` must implement are `associationsDo:`, `at:put`, and `removeKey:ifAbsent`. Iteration over associations can be delegated to the list of associations. Method `at:put:` searches for the association containing the given key. If it finds such an association, it mutate the association's value. If it finds no such association, it adds one.

S564c. ⟨*other methods of class* `Dictionary` S564c⟩≡ (S564b) S565a▷

```
(method associationsDo: (aBlock) (table do: aBlock))
(method at:put: (key value) [locals tempassoc]
    (set tempassoc (self associationAt:ifAbsent: key {}))
    ((tempassoc isNil) ifTrue:ifFalse:
         {(table add: (Association withKey:value: key value))}
         {(tempassoc setValue: value)})
    self)
```

When a key is removed, the associated value must first be saved, because the key-removal method is obligated to answer that value. The actual removal is done by sending the `reject:` message to the representation.

S565a. ⟨*other methods of class* `Dictionary` S564c⟩+≡ (S564b) ◁S564c S565b▷

```
(method removeKey:ifAbsent: (key exnBlock)
   [locals value-removed] ; value found if not absent
   (set value-removed (self at:ifAbsent: key {(return (exnBlock value))}))
   (set table (table reject: [block (assn) (key = (assn key))])) ; remove assoc
   value-removed)
```

Because more than one association might have the same value, it makes no sense to implement `remove:ifAbsent:`.

S565b. ⟨*other methods of class* `Dictionary` S564c⟩+≡ (S564b) ◁S565a S565c▷

```
(method remove:ifAbsent: (value exnBlock)
   (self error: 'Dictionary-uses-remove:key:-not-remove:))
```

And because a dictionary requires not just a value but also a key, the only sensible thing to add is an `Association`.

S565c. ⟨*other methods of class* `Dictionary` S564c⟩+≡ (S564b) ◁S565b S565d▷

```
(method add: (anAssociation)
  (self at:put: (anAssociation key) (anAssociation value)))
```

A dictionary's `print` method uses `associationsDo:`.

S565d. ⟨*other methods of class* `Dictionary` S564c⟩+≡ (S564b) ◁S565c

```
(method print () [locals print-comma]
    (set print-comma false)
    (self printName)
    (left-round print)
    (self associationsDo:
        [block (x) (space print)
                   (print-comma ifTrue: {(', print) (space print)})
                   (set print-comma true)
                   ((x key) print)   (space print)
                   ('|--> print)     (space print)
                   ((x value) print)])
    (space print)
    (right-round print)
    self)
```

U.5.8 Implementation of `Array`

In Smalltalk, arrays are one-dimensional and have a fixed size. The abstraction is a mutable sequence indexed with integer keys, starting from 0. The representation is primitive—an ML array. There is no representation invariant, and the abstraction function is essentially the identity function.

Many of `Array`'s methods are primitive, including array creation and the `at:`, `at:put:`, and `size` methods. These methods are defined in the interpreter, in chunks S554g–S555b in Section U.3.7.

S565e. ⟨*collection classes* S563⟩+≡ (S548d) ◁S564b

```
(class Array
    [subclass-of SequenceableCollection] ; representation is primitive
    (class-method new: (size) (primitive arrayNew self size))
    (class-method new  ()     (self error: 'size-of-Array-must-be-specified))
    (method size       ()     (primitive arraySize self))
    (method at:        (key)       (primitive arrayAt self key))
    (method at:put:    (key value) (primitive arrayUpdate self key value) self)
    (method printName  () nil) ; names of arrays aren't printed
    ⟨other methods of class Array 661c⟩
)
```

Since it's not useful to create an array without specifying a size, the `new` method on class `Array` reports an error.

An array is mutable, but it has a fixed size, so trying to add or remove an element is senseless. Because `add:` doesn't work, the inherited implementations of `select:` and `collect:` don't work either. Writing implementations that do work is Exercise 21 in Chapter 10.

S566a. ⟨*other methods of class* `Array` **⟦prototype⟧** S566a⟩≡

```
(method select:  (_) (self leftAsExercise))
(method collect: (_) (self leftAsExercise))
```

A working implementation of class method `withAll:` is also an exercise (Exercise 20, Chapter 10).

U.5.9 Remaining methods of `Number`: powers and roots

Numbers can be squared or raised to other integer powers. Method `squared` is easy. Method `raisedToInteger:` computes x^n using a standard algorithm that requires $O(\log n)$ multiplications. The algorithm has two base cases, for x^0 and x^1.

S566b. ⟨*other methods of class* `Number` S566b⟩≡ (663a) ◁663b S566c▷

```
(method squared () (self * self))
(method raisedToInteger: (anInteger)
    ((anInteger = 0) ifTrue:ifFalse:
        {(self coerce: 1)}
        {((anInteger = 1) ifTrue:ifFalse: {self}
            {(((self raisedToInteger: (anInteger div: 2)) squared) *
                (self raisedToInteger: (anInteger mod: 2)))})}))
```

Numbers can also have their square roots taken. My implementation uses Newton-Raphson iteration. Given input n, this algorithm uses an initial approximation $x_0 = 1$ and improves it stepwise. At step i, the improved approximation is $x_i = (x_{i-1} + n/x_{i-1})/2$. To know when to stop improving, the algorithm needs a *convergence condition*, which examines x_i and x_{i-1} and says when they are close enough to accept x_i as the answer.[1] My convergence condition is $|x_i - x_{i-1}| < \epsilon$. The default ϵ used in `sqrt` is $1/100$. Using `coerce:` ensures that the same `sqrt` method can be used for both fractions and floats.

S566c. ⟨*other methods of class* `Number` S566b⟩+≡ (663a) ◁S566b

```
(method sqrt () (self sqrtWithin: (self coerce: (1 / 100))))
(method sqrtWithin: (epsilon) [locals two x<i-1> x<i>]
    ; find square root of receiver within epsilon
    (set two    (self coerce: 2))
    (set x<i-1> (self coerce: 1))
    (set x<i>   ((x<i-1> + (self / x<i-1>)) / two))
    ({(((x<i-1> - x<i>) abs) > epsilon)} whileTrue:
          {(set x<i-1> x<i>)
           (set x<i> ((x<i-1> + (self / x<i-1>)) / two))})
    x<i>)
```

[1] The idea is that if $x_i \approx x_{i-1}$, $x_i = (x_{i-1} + n/x_{i-1})/2 \approx (x_i + n/x_i)/2$, and solving yields $x_i \approx \sqrt{n}$.

U.5.10 Implementation of integers

In addition to the Integer methods defined in Chapter 10 (page 664), integers also support a timesRepeat: method, which executes a loop a finite number of times.

S567a. ⟨*other methods of class* Integer S567a⟩≡ (664a) ◁669c

```
(method timesRepeat: (aBlock) [locals count]
    ((self isNegative) ifTrue: {(self error: 'negative-repeat-count)})
    (set count self)
    ({(count != 0)} whileTrue:
         {(aBlock value)
          (set count (count - 1))}))
```

The only concrete integer class built into μSmalltalk is SmallInteger. Almost all its methods are primitive. They are defined in chunks S553e–S554a.

S567b. ⟨*numeric classes* S567b⟩≡ (S548d) ◁670 S568a▷

```
(class SmallInteger
    [subclass-of Integer] ; primitive representation
    (class-method new: (n) (primitive newSmallInteger self n))
    (class-method new  ()  (self new: 0))
    (method negated    ()  (0 - self))
    (method print      ()  (primitive printSmallInteger self))
    (method +          (n) (primitive + self n))
    (method -          (n) (primitive - self n))
    (method *          (n) (primitive * self n))
    (method div:       (n) (primitive div self n))
    (method =          (n) (primitive sameObject self n))
    (method <          (n) (primitive < self n))
    (method >          (n) (primitive > self n))
)
```

The primitives don't support *mixed arithmetic,* e.g., comparison of integers and fractions. Writing better methods is a task you can do in Exercise 36 of Chapter 10.

U.5.11 Implementation of floating-point numbers

The original Smalltalk systems were built on the Xerox Alto, the world's first personal computer. Because the Alto had no hardware support for floating-point computation, floating-point computations were done in software. The implementation I present here would be suitable for such a machine (although more bits of precision in the mantissa would be welcome).

An object of class Float is an abstraction of a rational number. The representation is an integer m (the *mantissa*) combined with an integer e (the *exponent*), stored in instance variables mant and exp. The abstraction function maps this representation to the number $m \cdot 10^e$. Both m and e can be negative. The representation invariant guarantees that the absolute value of the mantissa is at most $2^{15} - 1$. The invariant ensures that two mantissas can be multiplied without overflow, even on an implementation that provides only 31-bit small integers.[2] The invariant is maintained with the help of a private normalize method: when a mantissa's magnitude exceeds $2^{15} - 1$, the normalize method divides the mantissa by 10 and increments the exponent until the mantissa is small enough. This operation loses precision; it is the source of so-called "floating-point rounding error." The possibility of rounding error implies that the answers obtained from floating-point arith-

[2] Some implementations of ML reserve one bit as a dynamic-type tag or as a tag for the garbage collector.

metic are approximate. This possibility is part of the specification of class `Float`, but specifying exactly what "approximate" means is beyond the scope of this book.

S568a. ⟨*numeric classes* S567b⟩+≡ (S548d) ◁S567b

```
(class Float
    [subclass-of Number]
    [ivars mant exp]
    (class-method mant:exp: (m e) ((self new) initMant:exp: m e))
    (method initMant:exp: (m e) ; private
        (set mant m) (set exp e) (self normalize))
    (method normalize ()    ; private
        ({((mant abs) > 32767)} whileTrue:
               {(set mant (mant div: 10))
                (set exp (exp + 1))})
        self)
    ⟨other methods of class Float S568b⟩
)
```

Like the other numeric classes, `Float` must provide methods that give a binary operation access to the representation of its argument.

S568b. ⟨*other methods of class* `Float` S568b⟩≡ (S568a) S568c ▷

```
(method mant () mant)  ; private
(method exp  () exp)   ; private
```

Comparing two floats with different exponents is awkward, so instead I compute their difference and compare it with zero.

S568c. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S568b S568d ▷

```
(method < (x) ((self - x) isNegative))
(method = (x) ((self - x) isZero))
```

Negation is easy: answer a new float with a negated mantissa.

S568d. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S568c S568e ▷

```
(method negated () (Float mant:exp: (mant negated) exp))
```

Method `negated`, together with the `+` method, also supports subtraction and comparison. Because of the way methods are inherited and work with one another, all the knowledge and effort required to add, subtract, or compare floating-point numbers with different exponents is captured in the `+` method. It's another victory for inheritance.

The `+` method adds $x' = m' \cdot 10^{e'}$ to `self`, which is $m \cdot 10^e$. Its implementation is based on the algebraic law $m \cdot 10^e = (m \cdot 10^{e-e'}) \cdot 10^{e'}$. This law implies

$$m \cdot 10^e + m' \cdot 10^{e'} = (m \cdot 10^{e-e'} + m') \cdot 10^{e'}.$$

I provide a naïve implementation which enforces $e - e' \geq 0$. This implementation risks overflow, but at least overflow can be detected. A naïve implementation using $e - e' \leq 0$ might well lose valuable bits of precision from m. A better implementation can be constructed using the ideas in Exercise 34.

S568e. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S568d S569a ▷

```
(method + (x-prime)
    ((exp >= (x-prime exp)) ifTrue:ifFalse:
        {(Float mant:exp: ((mant * (10 raisedToInteger: (exp - (x-prime exp)))) +
                           (x-prime mant))
                        (x-prime exp))}
        {(x-prime + self)}))
```

Multiplication is much simpler: $(m \cdot 10^e) \cdot (m' \cdot 10^{e'}) = (m \cdot m') \cdot 10^{e+e'}$. The product's mantissa $m \cdot m'$ may be large, but the class method `mant:exp:` normalizes it.

S569a. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S568e S569b▷

```
(method * (x-prime)
   (Float mant:exp: (mant * (x-prime mant)) (exp + (x-prime exp))))
```

I compute the reciprocal using the algebraic law

$$\frac{1}{m \cdot 10^e} = \frac{10^9}{m \cdot 10^9 \cdot 10^e} = \frac{10^9}{m} \cdot 10^{-e-9}.$$

Dividing 10^9 by m ensures the computation doesn't lose too much precision from m.

S569b. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S569a S569c▷

```
(method reciprocal ()
   (Float mant:exp: (1000000000 div: mant) (-9 - exp)))
```

Coercing converts to `Float`, and converting `Float` to `Float` is the identity.

S569c. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S569b S569d▷

```
(method coerce: (aNumber) (aNumber asFloat))
(method asFloat () self)
```

When converting a float to another class of number, a negative exponent means divide, and a nonnegative exponent means multiply.

S569d. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S569c S569e▷

```
(method asInteger ()
   ((exp isNegative) ifTrue:ifFalse:
      {(mant div: (10 raisedToInteger: (exp negated)))}
      {(mant    * (10 raisedToInteger: exp))}))
```

To get a fraction, I either put a power of 10 in the denominator, or I make a product with 1 in the denominator.

S569e. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S569d S569f▷

```
(method asFraction ()
   ((exp < 0) ifTrue:ifFalse:
      {(Fraction num:den: mant (10 raisedToInteger: (exp negated)))}
      {(Fraction num:den: (mant * (10 raisedToInteger: exp)) 1)}))
```

Unlike the sign tests in `Fraction`, the sign tests in `Float` aren't just an optimization: the < method sends `negative` to a floating-point number, so the superclass implementation of `negative`, which sends < to self, would lead to infinite recursion. Fortunately, the sign of a floating-point number is the sign of its mantissa, so all four methods can be delegated to `Integer`.

S569f. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S569e S570a▷

```
(method isZero             () (mant isZero))
(method isNegative         () (mant isNegative))
(method isNonnegative      () (mant isNonnegative))
(method isStrictlyPositive () (mant isStrictlyPositive))
```

A floating-point number is printed as m`x10^`e. But I want to avoid printing a number like 77 as `770x10^-1`. So if my `print` method sees a number with a negative exponent and a mantissa that is a multiple of 10, it divides the mantissa by 10 and increases the exponent, continuing until the exponent reaches zero or the mantissa is no longer a multiple of 10. As a result, a whole number always prints as a whole number times 10^0, no matter what its internal representation is.

S570a. ⟨*other methods of class* `Float` S568b⟩+≡ (S568a) ◁S569f

```
(method print ()
    (self print-normalize)
    (mant print) ('x10^ print) (exp print)
    (self normalize))

(method print-normalize ()
    ({((exp < 0) and: {((mant mod: 10) = 0)})} whileTrue:
        {(set exp (exp + 1)) (set mant (mant div: 10))}))
```

U.5.12 Implementation of compiled methods

A compiled method is just a box in which μSmalltalk code can be stored, for use as an argument to `setMethod`. It has no instance variables and answers no special messages.

S570b. ⟨*predefined μSmalltalk classes and values* S548d⟩+≡ ◁S561c

```
(class CompiledMethod
  [subclass-of Object]
)
```

S570c. ⟨*support for bootstrapping classes/values used during parsing* S561a⟩+≡ (S547) ◁S561a

```
local
  val compiledMethodClass = ref NONE : class option ref
in
  fun mkCompiledMethod m = (valOf (!compiledMethodClass), METHODV m)
    handle Option =>
      raise InternalError "Bad compiledMethodClass"
  fun saveCompiledMethodClass xi =
    compiledMethodClass := SOME (findClass ("CompiledMethod", xi))
end
```

U.6 Unit testing

Unit testing in μSmalltalk looks a little different from unit testing in μScheme or μML, but a little more like unit testing in Molecule: not only does testing for equality require a call to `eval`, but also printing is different. If a value needs to be printed, the testing code can't first convert it to a string, because in general, a μSmalltalk object doesn't know how to convert itself to a string. But a value *can* be asked to print itself, so when a value needs to be printed, the testing code sends it a `print` message. Values are printed by function `printsAs`, which sends a `print` message to an object and places the results in a buffer, the contents of which it then returns.

S571a. ⟨*definition of* `testIsGood` *for μSmalltalk* S571a⟩≡ (S548c)

```
fun testIsGood (test, xi) =
  let fun ev e = eval (e, emptyEnv, objectClass, noFrame, xi)
      fun outcome e = withHandlers (OK o ev) e (ERROR o stripAtLoc)
                      before resetTrace ()
      fun testEquals (v1, v2) =
        let val areSimilar = ev (SEND (nullsrc, VALUE v1, "=", [VALUE v2]))
        in  eqRep (areSimilar, mkBoolean true)
        end
      fun printsAs v =
        let val (bprint, contents) = bprinter ()
            val _ = withXprinter bprint ev (SEND (nullsrc, VALUE v, "print", [])
        in  contents ()
        end
      fun valueString _ =
        raise RuntimeError "internal error: called the wrong ValueString"
      ⟨definitions of check{Expect,Assert,Error{Passes that call printsAs S571b⟩
      ⟨definition of checkPrintPasses S573a⟩
      fun passes (CHECK_EXPECT (c, e)) = checkExpectPasses (c, e)
        | passes (CHECK_ASSERT c)      = checkAssertPasses c
        | passes (CHECK_ERROR c)       = checkErrorPasses  c
        | passes (CHECK_PRINT (c, s))  = checkPrintPasses  (c, s)
  in  passes test
  end
```

In case of a test failure, function `printsAs` is used to show what was expected.

S571b. ⟨*definitions of* `check{Expect,Assert,Error{Passes` *that call* `printsAs` S571b⟩≡ (S571a) S

```
fun whatWasExpected (LITERAL (NUM n), _) = printsAs (mkInteger n)
  | whatWasExpected (LITERAL (SYM x), _) = printsAs (mkSymbol x)
  | whatWasExpected (e, OK v) =
      concat [printsAs v, " (from evaluating ", expString e, ")"]
  | whatWasExpected (e, ERROR _) =
      concat ["the result of evaluating ", expString e]
```

Once printing is sorted out, the implementations of the unit-test forms are similar to the implementations of the same forms in other interpreters.

S572a. ⟨*definitions of* `check{Expect,Assert,Error{Passes` *that call* `printsAs` S571b⟩+≡ (S571a) ◁S571b S572

```
val cxfailed = "check-expect failed: "
fun checkExpectPasses (checkx, expectx) =
  case (outcome checkx, outcome expectx)
    of (OK check, OK expect) =>
         (case withHandlers (OK o testEquals)
                            (check, expect)
                            (ERROR o stripAtLoc)
            of OK true => true
             | OK false =>
                 failtest [cxfailed, "expected ", expString checkx,
                           " to be similar to ",
                           whatWasExpected (expectx, OK expect),
                           ", but it's ", printsAs check]
             | ERROR msg =>
                 failtest [cxfailed, "testing equality of ",
                           expString checkx, " to ",
                           expString expectx, " caused error ", msg])
     | (ERROR msg, tried) =>
         failtest [cxfailed, "evaluating ", expString checkx,
                   " caused error ", msg]
     | (_, ERROR msg) =>
         failtest  [cxfailed, "evaluating ", expString expectx,
                    " caused error ", msg]
```

S572b. ⟨*definitions of* `check{Expect,Assert,Error{Passes` *that call* `printsAs` S571b⟩+≡ (S571a) ◁S572a S572

```
val cafailed = "check-assert failed: "
fun checkAssertPasses checkx =
  case outcome checkx
    of OK check =>
         eqRep (check, mkBoolean true) orelse
         failtest [cafailed, "expected assertion ", expString checkx,
                   " to hold, but it doesn't"]
     | ERROR msg =>
         failtest [cafailed, "evaluating ", expString checkx,
                   " caused error ", msg]
```

S572c. ⟨*definitions of* `check{Expect,Assert,Error{Passes` *that call* `printsAs` S571b⟩+≡ (S571a) ◁S572b

```
val cefailed = "check-error failed: "
fun checkErrorPasses checkx =
      case outcome checkx
        of ERROR _ => true
         | OK check =>
             failtest [cefailed, "expected evaluating ", expString checkx,
                       " to cause an error, but evaluation produced ",
                       printsAs check]
```

Since the print methods are used everywhere—both in unit testing and in ordinary interaction—μSmalltalk provides a special unit-test form just for testing them. The check-print form has no analog in other interpreters.

S573a. ⟨*definition of* checkPrintPasses S573a⟩≡ (S571a)

```
val cpfailed = "check-print failed: "
fun checkPrintPasses (checkx, s) =
  case outcome checkx
    of OK check =>
         (case withHandlers (OK o printsAs) check (ERROR o stripAtLoc)
            of OK s' =>
                 s = s' orelse
                 failtest [cpfailed, "expected \"", s,
                           "\" but got \"", s', "\""]
             | ERROR msg =>
                 failtest [cpfailed, "calling print method on ",
                           expString checkx, " caused error ", msg])
     | ERROR msg =>
         failtest [cpfailed, "evaluating ", expString checkx,
                   " caused error ", msg]
```

U.7 String conversion

Function expString is analogous to what we see in other interpreters.

S573b. ⟨*definition of* expString *for μSmalltalk* S573b⟩≡ (S548a)

```
fun expString e =
  let fun bracket s = "(" ^ s ^ ")"
      val bracketSpace = bracket o spaceSep
      fun exps es = map expString es
      fun symString x = x
      fun valueString (_, NUM n) = intString n
        | valueString (_, SYM x) = "'" ^ symString x
        | valueString (c, _) = "<" ^ className c ^ ">"
  in  case e
        of LITERAL (NUM n) => intString n
         | LITERAL (SYM n) => "'" ^ symString n
         | LITERAL _ => "<wildly unexpected literal>"
         | VAR name => name
         | SET (x, e) => bracketSpace ["set", x, expString e]
         | RETURN e   => bracketSpace ["return", expString e]
         | SEND (_, e, msg, es) => bracketSpace (expString e :: msg :: exps es)
         | BEGIN es => bracketSpace ("begin" :: exps es)
         | PRIMITIVE (p, es) => bracketSpace ("primitive" :: p :: exps es)
         | BLOCK ([], es) => "[" ^ spaceSep (exps es) ^ "]"
         | BLOCK (xs, es) =>
             bracketSpace ["block", bracketSpace xs, spaceSep (exps es)]
         | METHOD (xs, [], es) =>
             bracketSpace ["compiled-method", bracketSpace xs,
                           spaceSep (exps es)]
         | METHOD (xs, ys, es) =>
             bracketSpace ["compiled-method", bracketSpace xs,
                           bracketSpace ("locals" :: ys), spaceSep (exps es)]
         | VALUE v => valueString v
         | SUPER => "super"
  end
```

Function valueString is *not* analogous to what we see in other interpreters. For example, valueString is never used in the read-eval-print loop; the read-eval-print loop prints values by sending them the print messages. Function valueString is used only when tracing message sends, because when you're trying to debug something, the last thing you want is for your debugging tool to send additional messages.

S574a. ⟨*definition of* valueString *for μSmalltalk* S574a⟩≡ (S548a)

```
fun valueString (c, NUM n) = intString n ^ valueString(c, USER [])
  | valueString (_, SYM v) = v
  | valueString (c, _) = "<" ^ className c ^ ">"
```

U.8 Lexical analysis and parsing

Lexical analysis and parsing are organized as follows:

S574b. ⟨*lexical analysis and parsing for μSmalltalk, providing* filexdefs *and* stringsxdefs S574b⟩≡ (S547)
⟨*lexical analysis for μSmalltalk* S574c⟩
⟨*parsers for single μSmalltalk tokens* S576b⟩
⟨*parsers and parser builders for formal parameters and bindings* (from chunk 697b)⟩
⟨*parsers and* xdef *streams for μSmalltalk* S575b⟩
⟨*shared definitions of* filexdefs *and* stringsxdefs S233a⟩

U.8.1 Lexical analysis

μScheme's lexer can't be reused for μSmalltalk, for two reasons: μSmalltalk treats curly braces as syntactic sugar for parameterless blocks, and μSmalltalk keeps track of source-code locations. Aside from these details, the lexers are the same.

The representation of a token is almost the same as in μScheme. But μSmalltalk accepts integer tokens that are too large to fit in a machine int, and μSmalltalk doesn't have the SHARP] token, because
in \usmalltalk, a [[# character does not introduce a Boolean.

S574c. ⟨*lexical analysis for μSmalltalk* S574c⟩≡ (S574b) S574d ▷

```
datatype pretoken = INTCHARS of char list
                  | NAME     of name
                  | QUOTE    of string option (* symbol or array *)
type token = pretoken plus_brackets
```

In error messages, a token may be converted back to a string.

S574d. ⟨*lexical analysis for μSmalltalk* S574c⟩+≡ (S574b) ◁ S574c S575a ▷

```
fun pretokenString (INTCHARS ds)    = implode ds
  | pretokenString (NAME    x)      = x
  | pretokenString (QUOTE NONE)     = "'"
  | pretokenString (QUOTE (SOME s)) = "'" ^ s
```

The lexer is similar to μScheme's lexer.

S575a. ⟨*lexical analysis for μSmalltalk* S574c⟩+≡ (S574b) ◁ S574d

smalltalkToken : token lexer

```
local
  val nondelims = many1 (sat (not o isDelim) one)

  fun validate NONE = NONE (* end of line *)
    | validate (SOME (#";", cs)) = NONE (* comment *)
    | validate (SOME (c, cs)) =
        let val msg = "invalid initial character in `" ^
                      implode (c::listOfStream cs) ^ "'"
        in  SOME (ERROR msg, EOS) : (pretoken error * char stream) option
        end
in
  val smalltalkToken =
    whitespace *> bracketLexer (
            (QUOTE o SOME o implode) <$> (eqx #"'" one *> nondelims)
        <|> QUOTE NONE               <$  eqx #"'" one
        <|> INTCHARS                 <$> intChars isDelim
        <|> (NAME o implode)         <$> nondelims
        <|> (validate o streamGet)
        )
end
```

U.8.2 Arity and colon checking for message names

Smalltalk has simple rules for computing the arity of a message based on the message's name: if the name is symbolic, the message is binary (one receiver, one argument); if the name is alphanumeric, the number of arguments is the number of colons. Unfortunately, in μSmalltalk a name can mix alphanumerics and symbols. Whether a message's name is considered symbolic or alphanumeric is determined by the name's *first* character.

S575b. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩≡ (S574b) S575c ▷

```
fun arity name =
      let val cs = explode name
          fun isColon c = (c = #":")
      in  if Char.isAlpha (hd cs) then
            length (List.filter isColon cs)
          else
            1
      end
```

Each message send is checked to see if the number of arguments matches the arity of the message name.

S575c. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁ S575b S576a ▷

```
fun arityOk name args = arity name = length args

fun arityErrorAt loc what msgname args =
  let fun argn n = if n = 1 then "1 argument" else intString n ^ " arguments"
  in  synerrorAt ("in " ^ what ^ ", message " ^ msgname ^ " expects " ^
                         argn (arity msgname) ^ ", but gets " ^
                         argn (length args)) loc
  end
```

To avoid confusion, μSmalltalk prohibits colons in symbolic message names. A method may have colons only if its first character is alphabetical. A message name must satisfy colonsOK or it is rejected.

S576a. ⟨*parsers and* xdef *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S575c S576c▷

```
fun colonsOK name =
  Char.isAlpha (String.sub (name, 0)) orelse not (Char.contains name #":")
  handle Subscript => false

val badColon = not o colonsOK

fun badColonErrorAt msgname loc =
  synerrorAt ("a symbolic method name like " ^ msgname ^
              " may not contain a colon")
             loc
```

U.8.3 Parsing

Parsers for tokens

The parser begins with parsers for individual tokens.

S576b. ⟨*parsers for single μSmalltalk tokens* S576b⟩≡ (S574b)

```
name : string parser
int  : int    parser
```

```
type 'a parser = (token, 'a) polyparser
val token : token parser = token (* make it monomorphic *)
val pretoken  = (fn (PRETOKEN t)=> SOME t  | _ => NONE) <$>? token
val namelike  = (fn (NAME s)         => SOME s  | _ => NONE) <$>? pretoken
val intchars  = (fn (INTCHARS ds)=> SOME ds | _ => NONE) <$>? pretoken
val sym   = (fn (QUOTE (SOME s)) => SOME s  | _ => NONE) <$>? pretoken
val quote = (fn (QUOTE NONE    ) => SOME () | _ => NONE) <$>? pretoken

val namelike = asAscii namelike

val int = intFromChars <$>! intchars
```

Parsers for expressions

μSmalltalk reserves a whole bunch of words, but not class. The word class can't be reserved because it's also the name of the message that is sent to any object to determine its class.

S576c. ⟨*parsers and* xdef *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S576a S577a▷

```
val reserved = [ "set", "begin", "primitive", "return", "block", "quote"
               , "compiled-method" , "subclass-of", "ivars", "method"
               , "class-method", "locals", "val", "define", "use"
               , "check-error", "check-expect", "check-assert"
               ]
val name = rejectReserved reserved <$>! namelike
```

In Smalltalk, the predefined "pseudovariables" `true`, `false`, `nil`, `self`, and `super` can't be mutated. Any attempt to assign to one of these pseudovariables is detected in the parser, so the fault can be identified without having to run the code. I define parsers for both `mutable` and `immutable` names.

S577a. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S576c S577b▷

```
mutable   : name parser
immutable : name parser
```

```
fun isImmutable x =
  List.exists (fn x' => x' = x) ["true", "false", "nil", "self", "super"]
val immutable = sat isImmutable name

val mutable =
  let fun can'tMutate (loc, x) =
        ERROR (srclocString loc ^
               ": you cannot set or val-bind pseudovariable " ^ x)
  in  can'tMutate <$>! @@ immutable <|> OK <$>! name
  end
```

If any μSmalltalk code tries to change any of the predefined "pseudovariables," the `mutable` parser causes an error.

An atomic expression is a numeric or symbolic literal, or a name.

S577b. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S577a S577c▷

```
val atomicExp
  =  LITERAL <$> NUM    <$> int
 <|> LITERAL <$> SYM    <$> (sym <|> (quote *> name)
                                 <|> (quote *> (intString <$> int)))
 <|> SUPER              <$  eqx "super" name
 <|> VAR                <$> name
```

A quoted literal is read by function `quotelit`, which in turn may call `mkSymbol`, `mkInteger`, or `mkArray`. These functions must not be called until after the initial basis is read in.

S577c. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S577b S578a▷

```
quotelit : value parser
```

```
fun quotelit tokens = (
         mkSymbol  <$> name
    <|>  mkInteger <$> int
    <|>  shaped ROUND  left <&> mkArray <$> bracket("(literal ...)", many quotel
    <|>  shaped SQUARE left <&> mkArray <$> bracket("(literal ...)", many quotel
    <|>  quote              <!> "' within ' is not legal"
    <|>  shaped CURLY  left  <!> "{ within ' is not legal"
    <|>  shaped CURLY  right <!> "} within ' is not legal"
    ) tokens
and shaped shape delim = sat (fn (_, s) => s = shape) delim
```

`<!>`	S261a
`<$>`	S249a
`<$>!`	S254a
`<$>?`	S252b
`<&>`	S252c
`<|>`	S249c
`asAscii`	S260b
`bracket`	S264a
`CURLY`	S256d
`eqx`	S252a
`ERROR`	S221b
`INTCHARS`	S574c
`intFromChars`	S256b
`intString`	S214c
`left`	S261c
`LITERAL`	688a
`many`	S253a
`mkArray`	698a
`mkInteger`	698a
`mkSymbol`	698a
`NAME`	S574c
`type name`	303
`NUM`	686a
`OK`	S221b
`type polyparser`	S259b
`PRETOKEN`	S257a
`QUOTE`	S574c
`rejectReserved`	S265b
`right`	S261c
`ROUND`	S256d
`sat`	S251c
`SQUARE`	S256d
`srclocString`	S233b
`SUPER`	688a
`SYM`	686a
`synerrorAt`	S235b
`type token`	S574c
`token`	S259c
`type value`	685
`VAR`	688a

Parsers for bracketed keyword expressions are similar to those in other bridge languages.

S578a. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S577c S578b▷

```
⟨definition of function bodyWarning S581c⟩
fun formalsIn context = formalsOf "(x1 x2 ...)" name context
fun sendClass (loc, e) = SEND (loc, e, "class", [])
val locals = usageParsers [("[locals y ...]", many name)] <|> pure []
fun method_body exp kind =
      (curry3 id <$> @@ (formalsIn kind) <*> locals <*> (bodyWarning <$> @@ (many exp)))
fun withoutArity f ((_, xs), ys, es) = f (xs, ys, es)

fun exptable exp = usageParsers
  [ ("(set x e)",             curry SET       <$> mutable <*> exp)
  , ("(begin e ...)",               BEGIN     <$> many exp)
  , ("(primitive p e ...)",   curry PRIMITIVE <$> name <*> many exp)
  , ("(return e)",                  RETURN    <$> exp)
  , ("(block (x ...) e ...)", curry BLOCK     <$> formalsIn "block" <*> many exp)
  , ("(compiled-method (x ...) [locals ...] e ...)",
                       withoutArity METHOD    <$> method_body exp "compiled method")
  , ("(class e)",                   sendClass <$> @@ exp)
  , ("(locals x ...)",
     pure () <!> "found '(locals ...)' where an expression was expected")
  , ("(quote v)",                   VALUE <$> quotelit)
  ]
```

Function `bodyWarning` is defined below.

If parser `exp` sees something it doesn't recognize, it can't result in an error—because it is used in `many exp`, it must simply fail. It does, however, recognize an empty message send, which is always incorrect.

S578b. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S578a S579a▷

`exp : exp parser`

```
⟨definition of function curlyWarning S581a⟩
fun exp tokens = (
      atomicExp
  <|> quote       *> (VALUE <$> quotelit) (* not while reading predefined classes *)
  <|> curlyBracket ("{exp ...}", curry BLOCK [] <$> curlyWarning <$> @@ (many exp))
  <|> exptable exp
  <|> liberalBracket ("(exp selector ...)",
                      messageSend <$> exp <*> @@ name <*>! many exp)
  <|> liberalBracket ("(exp selector ...)", noMsg <$>! @@ exp)
  <|> left *> right <!> "empty message send ()"
  )
  tokens
and noReceiver (loc, m) =
      synerrorAt ("sent message " ^ m ^ " to no object") loc
and noMsg (loc, e) =
      synerrorAt ("found receiver " ^ expString e ^ " with no message") loc
and messageSend receiver (loc, msgname) args =
      if badColon msgname then
          badColonErrorAt msgname loc
      else if arityOk msgname args then
          OK (SEND (loc, receiver, msgname, args))
      else
          arityErrorAt loc "message send" msgname args
```

Function `curlyWarning` is defined below.

Unit tests are recognized by `testtable`.

S579a. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S578b S579b▷

`testtable : unit_test parser`

```
val printable = name <|> implode <$> intchars

val testtable = usageParsers
  [ ("(check-expect e1 e2)", curry CHECK_EXPECT <$> exp <*> exp)
  , ("(check-assert e)",           CHECK_ASSERT <$> exp)
  , ("(check-error e)",            CHECK_ERROR  <$> exp)
  , ("(check-print e chars)", curry CHECK_PRINT <$> exp <*> printable)
  ]
```

Definitions of both class methods and instance methods are recognized by `method`.

S579b. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S579a S580a▷

`method : method_def parser`

```
val method =
  let fun method kind (nameloc, name) impl =
            check (kname kind, nameloc, name, impl) >>=+
            (fn (formals, locals, body) =>
                { flavor = kind, name = name, formals = formals
                , locals = locals, body = body })
      and kname IMETHOD = "method"
        | kname CMETHOD = "class-method"
      and check (kind, nameloc, name, (formals, locals, body)) =
            let val (formalsloc, xs) = formals
            in  if badColon name then
                    badColonErrorAt name nameloc
                else if arityOk name xs then
                  OK (xs, locals, BEGIN body)
                else
                  arityErrorAt formalsloc (kind ^ " definition") name xs
            end
      val mbody = method_body exp
  in  usageParsers
      [ ("(method f (args) body)",
                    method IMETHOD <$> @@ name <*>! mbody "method")
      , ("(class-method f (args) body)",
                    method CMETHOD <$> @@ name <*>! mbody "class method")
      ]
  end
val parseMethods = many method <* many eol <* eos
```

True definitions are recognized by `deftable`.

S580a. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S579b S580b▷

```
deftable : def parser
```

```
fun classDef name super ivars methods =
      CLASSD { name = name, super = super, ivars = ivars, methods = methods }

val ivars =
  nodups ("instance variable", "class definition") <$>! @@ (many name)

val subclass_of = usageParsers [("[subclass-of className]", name)]
val ivars = (fn xs => getOpt (xs, [])) <$>
            optional (usageParsers [("[ivars name...]", ivars)])

val deftable = usageParsers
  [ ("(val x e)", curry  VAL    <$> mutable <*> exp)
  , ("(define f (args) body)",
                 curry3 DEFINE <$> name <*> formalsIn "define" <*> exp)
  , ("(class name [subclass-of ...] [ivars ...] methods)",
                 classDef <$> name <*> subclass_of <*> ivars <*> many method
             <|> (EXP o sendClass) <$> @@ exp)

  ]
```

Extended definitions are recognized by `xdef`. The `xdef` parser recognizes `method` and `class-method`, because if a class definition has an extra right parenthesis, a `method` or `class-method` keyword might show up at top level.

S580b. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S580a S580c▷

```
xdeftable : xdef parser
xdef      : xdef parser
```

```
val xdeftable =
  let fun bad what =
        "unexpected `(" ^ what ^ "...'; " ^
        "did a class definition end prematurely?"
  in  usageParsers
      [ ("(use filename)",      USE <$> name)
      , ("(method ...)",        pzero <!> bad "method")
      , ("(class-method ...)",  pzero <!> bad "class-method")
      ]
  end

val xdef =  DEF  <$> deftable
        <|> TEST <$> testtable
        <|> xdeftable
        <|> badRight "unexpected right bracket"
        <|> DEF <$> EXP <$> exp
        <?> "definition"
```

S580c. ⟨*parsers and* `xdef` *streams for μSmalltalk* S575b⟩+≡ (S574b) ◁S580b

```
val xdefstream = interactiveParsedStream (smalltalkToken, xdef)
```

U.8.4 Warnings for common mistakes

It’s a common mistake to write a message send inside curly brackets but to forget the round brackets. If code contains a list of expressions and if any one except the last of those expressions is a name, this mistake has probably been made.

S581a. ⟨*definition of function* `curlyWarning` S581a⟩≡ (S578b)

```
  fun curlyWarning (loc, es) =  curlyWarning : exp list located -> exp list
    let fun nameFollowed (VAR _ :: _ :: _) = true
          | nameFollowed (_ :: es) = nameFollowed es
          | nameFollowed [] = false
        val () = if nameFollowed es then
                     warnAt loc ["inside {...} it looks like (...) was forgotten"]
                   else
                     ⟨warn about blocks of the form {exp name{ S581b⟩
    in  es
    end
```

It’s a little less definitive, but a block of the form {e $name$} also suggests that round brackets have been forgotten.

S581b. ⟨*warn about blocks of the form* `{exp name{` S581b⟩≡ (S581a)

```
  case es
    of [_, VAR method] =>
       let val method = "`" ^ method ^ "`"
       in  warnAt loc ["inside {...} it looks like ",
                       method, " was meant to send a ",
                       "message. If so, then wrap the message ",
                       "send in round brackets (...).  ",
                       "If the code is meant to answer ", method,
                       ", insert a literal `0` before ", method, "."]
       end
     | _=> ()
```

Similarly, the body of a method can be a list of expressions, but a message send is more common. If the list of expressions includes a variable or literal that is followed by something, it’s almost certain that round brackets have been forgotten, since otherwise the variable or literal would be evaluated only for its (nonexistent) side effect.

S581c. ⟨*definition of function* `bodyWarning` S581c⟩≡ (S578a)

```
  fun bodyWarning (loc, es) =
    let fun nameOrLitFollowed (VAR _     :: _ :: _) = true
          | nameOrLitFollowed (LITERAL _ :: _ :: _) = true
          | nameOrLitFollowed (_ :: es) = nameOrLitFollowed es
          | nameOrLitFollowed [] = false
         val () = if nameOrLitFollowed es then
                      warnAt loc ["it looks like the body of the method ",
                                  "should be wrapped in (...)"]
                    else
                      ()
    in  es
    end
```

U.9 Support for tracing

The μSmalltalk interpreter supports two forms of diagnostics that are called *tracing*:

- When requested by a program, *message tracing* shows every message send and the reply to it.
- When a checked run-time error occurs, *stack tracing* shows the state of the call stack, that is, every active send.

Both forms of tracing share some mutable state, and both are implemented in this section. To keep the details hidden from the rest of the interpreter, the shared mutable state and related functions are made `local`.

S582a. ⟨*functions for managing and printing a μSmalltalk stack trace* S582a⟩≡ (S548c)

```
local
  ⟨private state and functions for printing indented message traces S582b⟩
  ⟨private state and functions for maintaining a stack of source-code locations S584c⟩
in
  ⟨exposed message-tracing functions S583a⟩
  ⟨exposed stack-tracing functions S585a⟩
end
```

U.9.1 Message tracing

Interface to the programmer

When global variable `&trace` is set to a nonzero number, the μSmalltalk interpreter prints a trace of every message send and its reply. And at every send, `&trace` is decremented, so if a programmer wants to trace just the next N messages, they set `&trace` to N. This work is done by function `traceMe`. When the evaluator wants to know if it should trace a message send, it calls `traceMe`. When `&trace] is a nonzero number, [[traceMe` returns `true`; otherwise it returns `false`. And when `&trace` is nonzero, `traceMe` decrements `&trace`.

S582b. ⟨*private state and functions for printing indented message traces* S582b⟩≡ (S582a) S584a ▷

```
fun traceMe xi =                                  traceMe : value ref env -> bool
  let val count = find("&trace", xi)
  in  case !count
        of (c, NUM n) =>
            if n = 0 then false
            else ( count := (c, NUM (n - 1))
                 ; if n = 1 then (xprint "<trace ends>\n"; false) else true
                 )
         | _ => false
  end handle NotFound _ => false
```

Interface to the evaluator

Message tracing is implemented by function `trace`. Function `trace` is defined *inside* function `eval`, which grants it direct access to information like the source-code location of the send and the class of the receiver. Function `trace` manipulates the private state of the tracing facility using three functions: `traceIndent`, which traces sends; outdentTrace, which traces replies; and resetTrace, which resets indentation. Functions traceIndent and outdentTrace also cooperate to maintain the stack of active sends, which they do by calling push and pop.

S583a. ⟨*exposed message-tracing functions* S583a⟩≡ (S582a)

```
resetTrace  : unit -> unit
traceIndent : string * srcloc * string ->
                        value ref env -> (unit -> string list) -> unit
outdentTrace  :         value ref env -> (unit -> string list) -> unit
```

```
fun resetTrace ()       = (locationStack := []; tindent := 0)
fun traceIndent what xi = (push what; tracePrint INDENT_AFTER   xi)
fun outdentTrace     xi = (pop ();    tracePrint OUTDENT_BEFORE xi)
```

Functions traceIndent and outdentTrace are Curried; each returns a partial application of tracePrint. This partial application, when applied to its final argument, prints strings produced by that argument when and only when tracing is enabled.

The trace function itself is given an action with which to perform the send; action is run by applying it to the empty tuple. If tracing is enabled, trace emits two tracing messages: one before and one after running the action. The printing is done conditionally by traceIndent and outdentTrace, which also (unconditionally) maintain the stack of active sends.

S583b. ⟨*definition of function* trace S583b⟩≡ (689b)

```
trace : (unit -> value) -> value
```

```
fun trace action =
  let val (file, line) = srcloc
      val c  = className startingClass
      val objString = if String.isPrefix "class " c then c
                      else "an object of class " ^ c
      val () =
        traceIndent (msgname, (file, line), objString) xi (fn () =>
          [file, ", line ", intString line, ": ", "Sending message (",
           spaceSep (msgname :: map valueString vs), ")", " to ", objString])
      fun traceOut answer =
        answer before
        outdentTrace xi (fn () =>
           [file, ", line ", intString line, ": ",
            "(", spaceSep (valueString obj :: msgname :: map valueString vs), ")
            " = ", valueString answer])

      fun traceReturn r =
        ( outdentTrace xi (fn () =>
             [file, ", line ", intString line, ": ",
              "(", spaceSep (valueString obj :: msgname :: map valueString vs),
                " terminated by return"])
        ; raise Return r
        )

  in  traceOut (action ()) handle Return r => traceReturn r
  end
```

Conditional printing with indentation

To depict how many sends are active at any given moment, each message trace is indented by a number of spaces proportional to the number of active sends. The current indentation is maintained in local variable `tindent`. Function `indent` uses `tindent` to print the indentation.

S584a. ⟨*private state and functions for printing indented message traces* S582b⟩+≡ (S582a) ◁S582b S584b▷

`indent : int -> unit`

```
val tindent = ref 0
fun indent 0 = ()
  | indent n = (xprint "  "; indent (n-1))
```

Any actual printing is done by `tracePrint`, conditional on `traceMe` returning `true`. The argument `direction` of type `indentation` controls the adjustment of `indent`. So that each message is indented the same amount as its reply, the code outdents from the previous level *before* printing a message; it indents from the current level *after* printing a reply.

S584b. ⟨*private state and functions for printing indented message traces* S582b⟩+≡ (S582a) ◁S584a

```
datatype indentation = INDENT_AFTER | OUTDENT_BEFORE

fun tracePrint direction xi f =
    if traceMe xi then
      let val msg = f () (* could change tindent *)
      in  ( if direction = OUTDENT_BEFORE then tindent := !tindent - 1 else ()
          ; indent (!tindent)
          ; app xprint msg
          ; xprint "\n"
          ; if direction = INDENT_AFTER   then tindent := !tindent + 1 else ()
          )
      end
    else
        ()
```

U.9.2 Stack tracing

Each active send is identified by a message name, the sending location, and a string describing the receiver. The stack of active sends is stored in private variable `locationStack`. This stack is displayed when an error occurs.

S584c. ⟨*private state and functions for maintaining a stack of source-code locations* S584c⟩≡ (S582a)

```
val locationStack = ref [] : (string * srcloc * string) list ref
fun push info = locationStack := info :: !locationStack
fun pop () = case !locationStack
               of []     => raise InternalError "tracing stack underflows"
                | h :: t => locationStack := t
```

More precisely, the stack of active sends is displayed when an error message is printed. Function `eprintlnTrace`, defined below, prints an error message and shows the stack of active sends. For it to be called, it needs to be passed to `readEvalPrintWith`. That means that the μSmalltalk interpreter needs its own implementation of `runStream`. (The other interpreters all share the version defined in Appendix H, on page S240.)

S584d. ⟨*function* `runStream` *for μSmalltalk, which prints stack traces* S584d⟩≡ (S547)

`runStream : string -> TextIO.instream -> interactivity -> basis -> basis`

```
fun runStream inputName input interactivity basis =
  let val _ = setup_error_format interactivity
      val prompts = if prompts interactivity then stdPrompts else noPrompts
      val xdefs = filexdefs (inputName, input, prompts)
  in  readEvalPrintWith eprintlnTrace (xdefs, basis, interactivity)
  end
```

Showing the stack trace itself is somewhat trickier than you might think. The issue is that if a Smalltalk program suffers from infinite recursion, the stack will not overflow until it has several *thousand* active sends. And stack overflow is quite common; because of Smalltalk's dynamic dispatch, it is all too easy to write recursions that don't terminate. Moreover, such recursions often involve two, three, or more methods. When printing such a stack, the tracing functions condense repeated sequences of active sends. This operation is implemented by a team of several functions.

The fundamental operation used to condense a stack is to remove a repeated sequence from the beginning of a list of active sends. Function `removeRepeat` takes an argument n and a list xs, considers the first n elements of xs as a block, and removes as many copies of that block as it can. The number of copies removed is k, and `removeRepeat` satisfies the following algebraic law:

$$\texttt{removeRepeat}\ n\ xs = (k, ys, zs)$$
$$\text{where } xs = (ys)^k zs$$
$$\texttt{length}\ ys = n$$

An element of the list xs may by of any type that admits equality, which in Standard ML is written with a type variable `''a`.

S585a. ⟨*exposed stack-tracing functions* S585a⟩≡ (S582a) S585b ▷

```
removeRepeat : int -> ''a list -> int * ''a list * ''a list
```

```
fun removeRepeat 0 xs = (0, [], xs)
  | removeRepeat n xs =
      let val header = List.take (xs, n)
          fun count k xs =
            if (header = List.take (xs, n)) handle Subscript => false then
              count (k + 1) (List.drop (xs, n))
            else
              (k, header, xs)
      in  count 0 xs
      end handle Subscript => (0, [], xs)
```

Since `removeRepeat` says *how many* copies of length n it removed (that's k), it can be used to *search* for a repeat. Function `findRepeat` looks for an initial repeated sequence of length n or greater, and removes it. It stops after trying $n = 20$.

S585b. ⟨*exposed stack-tracing functions* S585a⟩+≡ (S582a) ◁ S585a S586a ▷

```
findRepeat : ''a list -> int -> int * ''a list * ''a list
```

```
fun findRepeat xs n =
  if n > 20 then
    (0, [], xs)
  else
    let val repeat as (k, _, _) = removeRepeat n xs
    in  if k >= 3 then
          repeat
        else
          findRepeat xs (n + 1)
    end
```

`type basis`	S548e
`eprintlnTrace`	S587a
`filexdefs`	S233a
`InternalError`	S219e
`noPrompts`	S268
`prompts`	S236c
`readEvalPrintWith`	S237
`setup_error_format`	S240a
`stdPrompts`	S268
`traceMe`	S582b
`xprint`	S215d

Repeated sequences don't always appear at the extreme young end of the stack. Sometimes when the stack is exhausted, there are a few active sends that aren't involved in the infinite recursion. Function `findRepeatAfter` searches for a repeated sequence *after* the first m or more elements of xs (the "header") are dropped. It tries m up to 10. It returns the header of length m, paired with whatever repeat is found by `findRepeat`.

S586a. ⟨*exposed stack-tracing functions* S585a⟩+≡ (S582a) ◁S585b S586b▷

```
findRepeatAfter : ''a list -> int -> ''a list * (int * ''a list * ''a list)
```

```
fun findRepeatAfter xs 10 = ([], (0, [], xs))
  | findRepeatAfter xs  m =
    let val (k, header, ys) = findRepeat (List.drop (xs, m)) 1
    in  if k > 0 then
          (List.take(xs, m), (k, header, ys))
        else
          findRepeatAfter xs (m + 1)
    end handle Subscript => ([], (0, [], xs))
```

To condense any given stack, `findRepeatAfter` is called with $m = 0$. The resulting information is stored in variable `headerAndRepeat`. If there's no reason to condense the stack, `headerAndRepeat` is initialized with an empty header and no repeats.

S586b. ⟨*exposed stack-tracing functions* S585a⟩+≡ (S582a) ◁S586a S587a▷

```
showStackTrace : bool -> unit
```

```
fun showStackTrace condense =
  if null (!locationStack) then
    ()
  else
    let fun showActiveSend (msg, (file, n), receiver) =
          app xprint ["  In ", file, ", line ", intString n,
                      ", sent `", msg, "' to ", receiver, "\n"]
        val headerAndRepeat =
          if condense then findRepeatAfter (!locationStack) 0
          else ([], (0, [], !locationStack))
        val _ = xprint "Method-stack traceback:\n"
    in  ⟨show the (possibly condensed) stack in headerAndRepeat S586c⟩
    end
```

The display of a stack depends on whether `findRepeatAfter` finds a header and a repeated sequence. If there's no header and no repeated sequence, all the active sends are shown. Otherwise the stack is shown in three parts: the header, the repeated sequence, and what's left.

S586c. ⟨*show the (possibly condensed) stack in* `headerAndRepeat` S586c⟩≡ (S586b)

```
case headerAndRepeat
  of ([], (0, _, locs)) => app showActiveSend locs
   | (_,  (0, _, _)) => raise InternalError "nonempty header with 0-length repeat"
   | (header, (k, repeated, locs)) =>
        ( app showActiveSend header
        ; if null header then ()
          else app xprint [ "    ... loop of size "
                          , Int.toString (length repeated) , " begins ...\n"
                          ]
        ; app showActiveSend repeated
        ; app xprint [ "    ... loop of size ", Int.toString (length repeated)
                     , " repeated ", Int.toString k, " times ...\n"
                     ]
        ; app showActiveSend locs
        )
```

A stack is condensed when some sort of resource is exhausted: either stack space ("recursion too deep") or the CPU cap.

S587a. ⟨*exposed stack-tracing functions* S585a⟩+≡ (S582a) ◁S586b

```
eprintlnTrace  : string -> unit
```

```
fun eprintlnTrace s =
  ( eprintln s
  ; showStackTrace (String.isSubstring "recursion too deep" s
                    orelse String.isSubstring "CPU time exhausted" s)
  ; resetTrace ()
  )
```

U.10 Other interpreter support

This final section of Appendix U defines a collection of boring utility functions that are used in Chapter 10, plus some specialized code for implementing `return` and for logging test coverage.

U.10.1 Utility functions

Optimization for the global environment

When a definition is evaluated, `evaldef` uses `optimizedBind` to add it to the global environment. Function `optimizedBind` is an optimized version of `bind`, just like the one used in Chapter 1. If a previous binding exists, it overwrites the previous binding and does not change the environment. The optimization is safe only because no operation in μSmalltalk makes a copy of the global environment.

S587b. ⟨*helper functions for evaluation* S587b⟩≡ (S548c)

```
fun optimizedBind (x, v, xi) =
  let val loc = find (x, xi)
  in  (loc := v; xi)
  end handle NotFound _ => bind (x, ref v, xi)
```

Utilities for error messages

If a block is sent a message with the wrong number of arguments, an error message is issued showing what message was expected. The message name (in Smalltalk lingo, "selector") is computed by function `valueSelector`.

S587c. ⟨*utility functions on μSmalltalk classes, methods, and values* S587c⟩≡ (S547) ◁686b S588a▷

```
fun valueSelector [] = "value"
  | valueSelector args = concat (map (fn _ => "value:") args)
```

Utilities for manipulating classes

Because a class can point to its superclass, the type `class` has to be a recursive type implemented as an ML `datatype`. So a value of type `class` is the value constructor `CLASS` applied to an ML record with a bunch of named fields. Pattern matching on such things is unpleasant, and when all I want is a class's name or its unique identifier, the notation is heavier than I would like. So I define two convenience functions. The "`...`" notation in each pattern match tells the Standard ML compiler that not all fields of the record in curly braces are mentioned, and the ones not mentioned should be ignored.

S588a. ⟨*utility functions on μSmalltalk classes, methods, and values* S587c⟩+≡ (S547) ◁ S587c S588b ▷

```
className : class -> name
classId   : class -> metaclass ref
```

```
fun className (CLASS {name,  ...}) = name
fun classId   (CLASS {class, ...}) = class
```

A method is also represented by a big record, and similar considerations apply. I extract a method's name using another convenience function, `methodName`. I also define `methodsEnv`, which builds an environment suitable for use in a class.

S588b. ⟨*utility functions on μSmalltalk classes, methods, and values* S587c⟩+≡ (S547) ◁ S588a S588e ▷

```
methodName : method -> name
methodsEnv : method list -> method env
```

```
fun methodName ({ name, ... } : method) = name
fun methodsEnv ms = foldl (fn (m, rho) => bind (methodName m, m, rho)) emptyEnv ms
```

When a new class object is defined, it inherits from a superclass. The interpreter needs not only the internal representation of the superclass, but also the internal representation of the superclass's metaclass. Both are found by function `findClassAndMeta`. If an object is found that is *not* a class, then somebody has tried to inherit from something that's not a class, which is a checked run-time error.

S588c. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S552a⟩+≡ (S550c) ◁ S559a S589b ▷

```
findClassAndMeta : name * value ref env -> class * class
```

```
fun findClassAndMeta (supername, xi) =
  case !(find (supername, xi))
    of (meta, CLASSREP c) => (c, meta)
     | v => raise RuntimeError ("object " ^ supername ^ " = " ^ valueString v ^
                                " is not a class")
```

As described in Chapter 10 (page 694), Smalltalk's classes point to their metaclasses, and some metaclasses point (indirectly) back to classes. My interpreter deals with the circularity by initializing a class's metaclass field to `PENDING`, then updated later. The update is performed by function `setMeta`, which insists that no metaclass field be updated more than once.

S588d. ⟨*metaclass utilities* S588d⟩≡ (S550c) ◁ 695b

```
fun setMeta (CLASS { class = m as ref PENDING, ... }, meta) = m := META meta
  | setMeta (CLASS { class = ref (META _), ... }, _) =
      raise InternalError "double patch"
```

Utilities for making new classes

In general, a new class is made by `mkClass`, which checks to be sure that no instance variable is repeated. Each class is identified by its `class` field, which points to a unique mutable location.

S588e. ⟨*utility functions on μSmalltalk classes, methods, and values* S587c⟩+≡ (S547) ◁ S588b

```
mkClass : name -> metaclass -> class -> name list -> method list -> class
```

```
fun mkClass name meta super ivars ms =
  ( ⟨if any name in ivars repeats a name declared in a superclass, raise RuntimeError S589a⟩
  ; CLASS { name = name, super = SOME super, ivars = ivars
          , methods = ref (methodsEnv ms), class = ref meta }
  )
```

The check for duplicate instance variables has to climb up the class hierarchy.

S589a. ⟨*if any name in* `ivars` *repeats a name declared in a superclass, raise* `RuntimeError` S589a⟩≡ (S588e)

```
let fun checkDuplicateIvar (SOME (CLASS { name = c', ivars, super, ... })) x =
        if member x ivars then
          raise RuntimeError ("Instance variable " ^ x ^ " of class " ^ name ^
                              " duplicates a variable of superclass " ^ c')
        else
          checkDuplicateIvar super x
    | checkDuplicateIvar NONE x = ()
in  app (checkDuplicateIvar (SOME super)) ivars
end
```

When a new class is created, its method definitions are converted to methods by function `methodDefns`. This function also separates class methods from instance methods. Each method has to know what `super` means; it means one thing in an instance method and another thing in a class method. Function `methodDefns` is organized like this:

- Information about superclasses is passed in: argument `super` is the superclass from which the new class inherits; `superMeta` is `super`'s metaclass.
- Internal function `method` builds the representation of a method from its syntax. It associates class `super` with each instance method and `superMeta` with each class method. These associations guarantee that every message sent to `SUPER` arrives at the proper destination.
- Internal function `addMethodDefn` processes each method definition, adding it either to the list of class methods or to the list of instance methods for the new class. To accumulate these lists and place them in `imethods` and `cmethods`, `methodDefns` applies `foldr` to `addMethodDefn`, a pair of empty lists, and the list of method definitions `ms`.

S589b. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S552a⟩+≡ (S550c) ◁S588c

```
methodDefns : class * class -> method_def list -> method list * method list
method : method_def -> method
```

```
fun methodDefns (superMeta, super) ms =
  let fun method { flavor, name, formals, locals, body } =
            { name = name, formals = formals, body = body, locals = locals
            , superclass = case flavor of IMETHOD => super
                                        | CMETHOD => superMeta
            }
      fun addMethodDefn (m as { flavor = CMETHOD, ... }, (c's, i's)) =
                                                  (method m :: c's, i's)
        | addMethodDefn (m as { flavor = IMETHOD, ... }, (c's, i's)) =
                                                  (c's, method m :: i's)
  in  foldr addMethodDefn ([], []) ms
  end
```

U.10.2 Frames and `return`

According to μSmalltalk's operational semantics, every message send allocates a new frame $\hat{F}$, by the predicate $\hat{F} \notin \mathcal{F}_n$. In the interpreter, a frame is represented by a value of type `frame`, each of which carries a unique integer. A new frame is allocated by function `newFrame`.

S590a. ⟨*support for μSmalltalk stack frames* S590a⟩≡ (S548a) S590b ▷

```
datatype frame = FRAME_NUMBER of int
local
  val next_f = ref 0
in
  fun newFrame () = FRAME_NUMBER (!next_f) before next_f := !next_f + 1
end
```

Top-level expressions and expressions in unit tests are evaluated outside the context of any message send. In these circumstances, `eval` is told that the current frame is `noFrame`.

S590b. ⟨*support for μSmalltalk stack frames* S590a⟩+≡ (S548a) ◁ S590a S590c ▷

```
val noFrame = newFrame () (* top level, unit tests, etc... *)
```

A μSmalltalk `return` is implemented by the ML `Return` exception. The exception keeps track of the frames that are unwound by the `return`; each element of the list has type `active_send`.

S590c. ⟨*support for μSmalltalk stack frames* S590a⟩+≡ (S548a) ◁ S590b S590f ▷

```
type active_send = { method : name, class : name, loc : srcloc }
```

Unwound frames are added to the list when `eval` catches the `Return` exception and unwinds the frame `this`.

S590d. ⟨*reraise* `Return`*, adding* `msgname`*,* `class`*, and* `loc` *to* `unwound` S590d⟩≡ (689b)

```
let val this = { method = msgname, class = className class, loc = srcloc }
in  raise Return { value = v, to = F', unwound = this :: unwound }
end
```

If a `return` is evaluated after the frame it refers to has died, all the frames are unwound, the interpreter issues an error message, and it prints all the unwound frames.

S590e. ⟨*handle unexpected* `Return` *in* `evaldef` S590e⟩≡ (693c)

```
handle Return { value = v, unwound = unwoundFrames, ... } =>
  if null unwoundFrames then
    raise RuntimeError
      ("tried to (return " ^ valueString v ^ ") from an activation that has died")
  else
    raise RuntimeError ("tried to return from an activation that has died:\n  " ^
                        String.concatWith "\n  " (map activeSendString unwoundFrames))
```

An unwound frame is printed by function `activeSendString`.

S590f. ⟨*support for μSmalltalk stack frames* S590a⟩+≡ (S548a) ◁ S590c

```
fun activeSendString { method, class, loc = (file, line) } =
  let val obj = if String.isPrefix "class " class then class
                else "an object of class " ^ class
  in  concat [file, ", line ", intString line, ": ", "sent `", method, "` to ", obj]
  end
```

Appendix V contents

V

Supporting code for μProlog

V.1 Organizing code chunks into an interpreter

Unlike the other bridge languages, μProlog does not evaluate syntax to produce values. Instead it substitutes terms for logical variables. Its overall structure is therefore different:

S593a. ⟨*upr.sml* S593a⟩≡
⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S213a⟩
⟨*abstract syntax for μProlog* S55c⟩
⟨*support for tracing μProlog computation* S609c⟩
⟨*substitution and unification* S78b⟩
⟨*renaming μProlog variables* S79b⟩
⟨*lexical analysis and parsing for μProlog, providing* `xdefsInMode` S598b⟩
⟨*evaluation, testing, and the read-eval-print loop for μProlog* S593b⟩
⟨*function* `runAs` *for μProlog* S609a⟩
⟨*code that looks at μProlog's command-line arguments and calls* `runAs` S609b⟩

The evaluation parts are organized as follows:

S593b. ⟨*evaluation, testing, and the read-eval-print loop for μProlog* S593b⟩≡ (S593a)
⟨*μProlog's database of clauses* S78a⟩
⟨*functions* `eval`*,* `is`*, and* `compare`*, used in primitive predicates* S594b⟩
⟨*tracing functions* S111⟩
⟨*search* (left as an exercise)⟩
⟨*interaction* S81c⟩
⟨*shared definition of* `withHandlers` S239a⟩
⟨*definitions of* `basis` *and* `processDef` *for μProlog* S81a⟩
⟨*shared unit-testing utilities* S225a⟩
⟨*definition of* `testIsGood` *for μProlog* S595a⟩
⟨*shared definition of* `processTests` S226⟩
⟨*shared read-eval-print loop* S237⟩

V.2 Primitives

The μProlog interpreter doesn't store a persistent initial basis in an ML variable. Instead, μProlog's primitive predicates sit in an environment that is used in the `query` function. These predicates are defined in this section, starting with `true`.

S593c. ⟨*μProlog's primitive predicates* `::` S593c⟩≡ (S80) S593d ▷

```
("true", fn args => fn succ => fn fail =>
          if null args then succ idsubst fail else fail ()) ::
```

Predicate `atom` tests to see if its argument is an atom.

S593d. ⟨*μProlog's primitive predicates* `::` S593c⟩+≡ (S80) ◁ S593c S594a ▷

```
("atom", fn args => fn succ => fn fail =>
           case args of [APPLY(f, [])] => succ idsubst fail
                      | _ => fail ()) ::
```

Printing a term always succeeds, and it produces the identity substitution.

S594a. ⟨*μProlog's primitive predicates* :: S593c⟩+≡ (S80) ◁S593d S594d▷

```
  ("print", fn args => fn succ => fn fail =>
              ( app (fn x => (print (termString x); print " ")) args
              ; print "\n"
              ; succ idsubst fail
              )) ::
```

Primitive predicate is requires a very small evaluator. Because it works only with integers, never with variables, the evaluator doesn't need an environment.

S594b. ⟨*functions* eval*,* is*, and* compare*, used in primitive predicates* S594b⟩≡ (S593b) S594c▷

eval : term -> int

```
  fun eval (LITERAL n) = n
    | eval (APPLY ("+", [x, y])) = eval x  +  eval y
    | eval (APPLY ("*", [x, y])) = eval x  *  eval y
    | eval (APPLY ("-", [x, y])) = eval x  -  eval y
    | eval (APPLY ("/", [x, y])) = eval x div eval y
    | eval (APPLY ("-", [x]))    = 0 - eval x
    | eval (APPLY (f, _))        =
        raise RuntimeError (f ^ " is not an arithmetic predicate " ^
                            "or is used with wrong arity")
    | eval (VAR v) = raise RuntimeError ("Used uninstantiated variable " ^ v ^
                                         " in arithmetic expression")
```

Predicate x is e evaluates term e as an integer expression and constrains it to equal x.

S594c. ⟨*functions* eval*,* is*, and* compare*, used in primitive predicates* S594b⟩+≡ (S593b) ◁S594b S594e▷

```
  fun is [x, e] succ fail = (succ (solve (x ~ LITERAL (eval e))) fail
                             handle Unsatisfiable => fail())
    | is _      _      fail = fail ()
```

S594d. ⟨*μProlog's primitive predicates* :: S593c⟩+≡ (S80) ◁S594a S594f▷

```
  ("is", is) ::
```

A comparison predicate is applied to exactly two arguments. If these arguments aren't integers, it's a run-time error. If they are, ML function cmp determines the success or failure of the predicate.

S594e. ⟨*functions* eval*,* is*, and* compare*, used in primitive predicates* S594b⟩+≡ (S593b) ◁S594c

```
  fun compare name cmp [LITERAL n, LITERAL m] succ fail =
        if cmp (n, m) then succ idsubst fail else fail ()
    | compare name _ [_, _] _ _ =
        raise RuntimeError ("Used comparison " ^ name ^ " on non-integer term")
    | compare name _ _ _ _ =
        raise InternalError ("this can't happen---non-binary comparison?!")
```

There are four comparison predicates.

S594f. ⟨*μProlog's primitive predicates* :: S593c⟩+≡ (S80) ◁S594d S594g▷

```
  ("<",  compare "<"  op < ) ::
  (">",  compare ">"  op > ) ::
  ("=<", compare "=<" op <= ) ::
  (">=", compare ">=" op >= ) ::
```

Each predicate above takes as argument a list of terms, a success continuation, and a failure continuation. Two more predicates, ! and not, cannot be implemented using this technique; they have to be added directly to the interpreter (Exercises 44 and 45). This code ensures that they can't be used by mistake.

S594g. ⟨*μProlog's primitive predicates* :: S593c⟩+≡ (S80) ◁S594f

```
  ("!",   fn _ => raise LeftAsExercise "the cut (!)") ::
  ("not", fn _ => raise LeftAsExercise "predicate `not'") ::
```

V.3 Unit testing

Unit testing in μProlog is different from any other unit testing, because the unit tests are different. Instead of comparing values or types, a unit test can check for satisfiability, and when given an explicit substitution, a test can check that the substitution satisfies the given query.

S595a. ⟨*definition of* `testIsGood` *for μProlog* S595a⟩≡ (S593b)

```
testIsGood : unit_test * basis -> bool
```

```
fun testIsGood (test, database) =
  let ⟨definitions of checkSatisfiedPasses and checkUnsatisfiablePasses S595b⟩
      fun passes (CHECK_UNSATISFIABLE gs)      = checkUnsatisfiablePasses gs
        | passes (CHECK_SATISFIABLE   gs)      = checkSatisfiablePasses gs
        | passes (CHECK_SATISFIED (gs, theta)) = checkSatisfiedPasses (gs, theta)
  in  passes test
  end
```

If a query fails a test, it is printed using function `qstring`.

S595b. ⟨*definitions of* `checkSatisfiedPasses` *and* `checkUnsatisfiablePasses` S595b⟩≡ (S595a) S595c ▷

```
type query
qstring : query -> string
```

```
type query = goal list
val qstring =
  nullOrCommaSep "?" o map goalString
```

All three unit tests work by passing appropriate success and failure continuations to `query`. To pass the `check-unsatisfiable` test, the query must be unsatisfiable. If the test fails, the satisfying substitution is shown without logical variables that are introduced by renaming clauses. Such variables begin with underscores, and they are removed by function `stripSubst`.

S595c. ⟨*definitions of* `checkSatisfiedPasses` *and* `checkUnsatisfiablePasses` S595b⟩+≡ (S595a

```
fun stripSubst theta =
  List.filter (fn (x, _) => String.sub (x, 0) <> #"_") theta
fun checkUnsatisfiablePasses (gs) =
  let fun succ theta' _ =
        failtest ["check_unsatisfiable failed: ", qstring gs,
                  " is satisfiable with ", substString theta']
      fun fail () = true
  in  query database gs (succ o stripSubst) fail
  end
```

To pass the `check-satisfiable` test, the query must be satisfiable.

S595d. ⟨*definitions of* `checkSatisfiedPasses` *and* `checkUnsatisfiablePasses` S595b⟩+≡ (S595a

```
fun checkSatisfiablePasses (gs) =
  let fun succ _ _ = true
      fun fail () = failtest ["check_unsatisfiable failed: ", qstring gs,
                              " is not satisfiable"]
  in  query database gs succ fail
  end
```

The check-satisfied test has an explicit substitution θ, and if that substitution has no logical variables, the test passes only if the query $\theta(gs)$ is satisfied by the identity substitution. (Logical variables introduced by renaming don't count.) If θ includes logical variables, $\theta(gs)$ merely has to be satisfiable.

S596a. ⟨*definitions of* checkSatisfiedPasses *and* checkUnsatisfiablePasses S595b⟩+≡ (S595a) ◁S595d

```
fun checkSatisfiedPasses (gs, theta) =
  let val thetaVars =
        foldl (fn ((_, t), fv) => union (termFreevars t, fv)) emptyset theta
      val ground = null thetaVars
      val gs' = map (goalsubst theta) gs
      fun succ theta' _ =
        if ground andalso not (null theta') then
          failtest ["check_satisfied failed: ", qstring gs,
                    " required additional substitution ", substString theta']
        else
          true
      fun fail () =
        failtest ["check_satisfied failed: could not prove ", qstring gs']
  in  query database gs' (succ o stripSubst) fail
  end
```

V.4 Substitution

The substitutions used in μProlog closely resemble the ones used in ML type inference (Chapter 7). A substitution θ is a structure-preserving mapping from terms to terms. As in Chapter 7, a substitution is represented as an environment. In μProlog, the environment maps logical variables to terms. All the substitution functions resemble the functions used to substitute types for type variables in Chapter 7.

S596b. ⟨*substitutions for μProlog* S596b⟩≡ (S78b) S596c ▷

```
type subst
idsubst : subst
```

```
type subst = term env
val idsubst = emptyEnv
```

A substitution is applied to a variable by varsubst.

S596c. ⟨*substitutions for μProlog* S596b⟩+≡ (S78b) ◁S596b S596d ▷

```
varsubst : subst -> (name -> term)
```

```
fun varsubst theta =
  (fn x => find (x, theta) handle NotFound _ => VAR x)
```

A substitution is applied to a term by termsubst.

S596d. ⟨*substitutions for μProlog* S596b⟩+≡ (S78b) ◁S596c S596e ▷

```
termsubst : subst -> (term -> term)
```

```
fun termsubst theta =
  let fun subst (VAR x)         = varsubst theta x
        | subst (LITERAL n)     = LITERAL n
        | subst (APPLY (f, ts)) = APPLY (f, map subst ts)
  in  subst
  end
```

A substitution is applied to a goal or a clause by goalsubst or clausesubst.

S596e. ⟨*substitutions for μProlog* S596b⟩+≡ (S78b) ◁S596d S597a ▷

```
goalsubst   : subst -> (goal   -> goal)
clausesubst : subst -> (clause -> clause)
```

```
fun goalsubst   theta (f, ts)   = (f, map (termsubst theta) ts)
fun clausesubst theta (c :- ps) = (goalsubst theta c :- map (goalsubst theta) ps)
```

And a substitution is applied to a constraint by consubst.

S597a. ⟨*substitutions for μProlog* S596b⟩+≡ (S78b) ◁S596e S597b▷

```
consubst : subst -> (con -> con)
```

```
fun consubst theta =
  let fun subst (t1 ~  t2) = termsubst theta t1 ~ termsubst theta t2
        | subst (c1 /\ c2) = subst c1 /\ subst c2
        | subst TRIVIAL    = TRIVIAL
  in  subst
  end
```

Substitutions are created using the same infix operator as in Chapter 7.

S597b. ⟨*substitutions for μProlog* S596b⟩+≡ (S78b) ◁S597a S597c▷

```
|--> : name * term -> subst
```

```
infix 7 |-->
fun x |--> (VAR x') = if x = x' then idsubst else bind (x, VAR x', emptyEnv)
  | x |--> t        = if member x (termFreevars t) then
                        raise InternalError "non-idempotent substitution"
                      else
                        bind (x, t, emptyEnv)
```

And substitutions compose just as in Chapter 7.

S597c. ⟨*substitutions for μProlog* S596b⟩+≡ (S78b) ◁S597b

```
compose : subst * subst -> subst
```

```
fun dom theta =
  map (fn (a, _) => a) theta
fun compose (theta2, theta1) =
  let val domain  = union (dom theta2, dom theta1)
      val replace = termsubst theta2 o varsubst theta1
  in  map (fn a => (a, replace a)) domain
  end
```

V.5 String conversions

This code converts terms, goals, and clauses to strings.

S597d. ⟨*definitions of* termString, goalString, *and* clauseString S597d⟩≡ (S55c) S597e▷

```
fun termString (APPLY ("cons", [car, cdr])) =
      let fun tail (APPLY ("cons", [car, cdr])) = ", " ^ termString car ^ tail c
            | tail (APPLY ("nil",  []))         = "]"
            | tail x                           = "|" ^ termString x ^ "]"
      in  "[" ^ termString car ^ tail cdr
      end
  | termString (APPLY ("nil", [])) = "[]"
  | termString (APPLY (f, []))     = f
  | termString (APPLY (f, [x, y])) =
      if Char.isAlpha (hd (explode f)) then appString f x [y]
      else String.concat ["(", termString x, " ", f, " ", termString y, ")"]
  | termString (APPLY (f, h::t)) = appString f h t
  | termString (VAR v) = v
  | termString (LITERAL n) = intString n
and appString f h t =
      String.concat (f :: "(" :: termString h ::
                     foldr (fn (t, tail) => ", " :: termString t :: tail) [")"]
```

S597e. ⟨*definitions of* termString, goalString, *and* clauseString S597d⟩+≡ (S55c) ◁S597d S598

```
fun goalString g = termString (APPLY g)
fun clauseString (g :- []) = goalString g
  | clauseString (g :- (h :: t)) =
      String.concat (goalString g :: " :- " :: goalString h ::
                     (foldr (fn (g, tail) => ", " :: goalString g :: tail)) [] t)
```

S598a. ⟨*definitions of* termString, goalString, *and* clauseString S597d⟩+≡ (S55c) ◁S597e

```
fun substString pairs =
      nullOrCommaSep "no substitution"
      (map (fn (x, t) => x ^ " = " ^ termString t) pairs)
```

V.6 Lexical analysis

The lexical-analysis code is structured as follows:

S598b. ⟨*lexical analysis and parsing for μProlog, providing* xdefsInMode S598b⟩≡ (S593a) S601b ▷

```
⟨lexical analysis for μProlog S598c⟩
local
  ⟨character-classification functions for μProlog S599a⟩
  ⟨lexical utility functions for μProlog S599d⟩
in
  ⟨lexical analyzers for for μProlog S600c⟩
end
```

V.6.1 Tokens

μProlog has a more complex lexical structure than other languages, so it has more forms of tokens. It has uppercase, lowercase, and symbolic tokens, as well as integers. And to simplify the parser, I distinguish reserved words and symbols using RESERVED. Because μProlog doesn't use the same bracketed-keyword syntax as the other bridge languages, I don't bother with the bracket machinery of Appendix I—a bracket is just another RESERVED token.

Finally, because a C-style μProlog comment can span multiple lines, the lexical analyzer may consume more than one line—in which process it may encounter the end of a file. Reading the end of file needs to be distinguishable from failing to read a token, so I represent end of file by its own special token EOF.

S598c. ⟨*lexical analysis for μProlog* S598c⟩≡ (S598b) S598d ▷

type token

```
datatype token
  = UPPER     of string
  | LOWER     of string
  | SYMBOLIC  of string
  | INT_TOKEN of int
  | RESERVED  of string
  | EOF
```

Tokens can be printed in error messages.

S598d. ⟨*lexical analysis for μProlog* S598c⟩+≡ (S598b) ◁S598c S599b ▷

```
fun tokenString (UPPER s)     = s
  | tokenString (LOWER s)     = s
  | tokenString (INT_TOKEN n) = intString n
  | tokenString (SYMBOLIC  s) = s
  | tokenString (RESERVED  s) = s
  | tokenString EOF           = "<end-of-file>"
```

V.6.2 Classification of characters

Just as μML distinguishes the names of value constructors from the names of value variables, μProlog distinguishes symbolic names (like +) from alphanumeric names (like add1). The distinction is established by distinguishing characters: every character is either a symbol, an alphanumeric, a space, or a delimiter.

S599a. ⟨*character-classification functions for μProlog* S599a⟩≡ (S598b)

```
val symbols = explode "!%^&*-+:=|~<>/?`$\\"
fun isSymbol c = List.exists (fn c' => c' = c) symbols
fun isIdent  c = Char.isAlphaNum c orelse c = #"_"
fun isSpace  c = Char.isSpace c
fun isDelim  c = not (isIdent c orelse isSymbol c)
```

V.6.3 *Reserved words and anonymous variables*

Tokens are formed from symbols or from lower-case letters by applying functions symbolic and lowers. Function symbolic usually returns a SYMBOLIC tokens, and lower usually returns a LOWER token, but there are exceptions. One exception is for reserved words. The other is for the cut: because the cut is nullary, not binary, it is treated as LOWER, just like any other nullary predicate.

S599b. ⟨*lexical analysis for μProlog* S598c⟩+≡ (S598b) ◁S598d S599c▷

```
symbolic : string -> token
lower    : string -> token
```

```
fun symbolic ":-" = RESERVED ":-"
  | symbolic "."  = RESERVED "."
  | symbolic "|"  = RESERVED "|"
  | symbolic "!"  = LOWER "!"
  | symbolic s    = SYMBOLIC s
fun lower "is" = RESERVED "is"
  | lower "check_satisfiable"   = RESERVED "check_satisfiable"
  | lower "check_unsatisfiable" = RESERVED "check_unsatisfiable"
  | lower "check_satisfied"     = RESERVED "check_satisfied"
  | lower s    = LOWER s
```

A variable consisting of a single underscore gets converted to a unique "anonymous" variable.

S599c. ⟨*lexical analysis for μProlog* S598c⟩+≡ (S598b) ◁S599b

```
fun anonymousVar () =
  case freshVar ""
    of VAR v => UPPER v
     | _ => raise InternalError "\"fresh variable\" is not a VAR"
```

V.6.4 *Converting characters to tokens*

Utility functions underscore and int make sure that an underscore or a sequence of digits, respectively, is never followed by any character that might be part of an alphanumeric identifier. When either of these functions succeeds, it returns an appropriate token. Each function takes two inputs: an argument and the sequence of characters that follow the argument in the input stream.

S599d. ⟨*lexical utility functions for μProlog* S599d⟩≡ (S598b) S600a▷

```
underscore : char      -> char list -> token error
int        : char list -> char list -> token error
```

```
fun underscore _ [] = OK (anonymousVar ())
  | underscore c cs = ERROR ("name may not begin with underscore at " ^
                                implode (c::cs))

fun int cs [] = intFromChars cs >>=+ INT_TOKEN
  | int cs ids =
      ERROR ("integer literal " ^ implode cs ^
             " may not be followed by '" ^ implode ids ^ "'")
```

>>=+	S222b
ERROR	S221b
type error	S221b
freshVar	S79b
InternalError	S219e
intFromChars	S256b
intString	S214c
nullOrCommaSep	S214f
OK	S221b
termString	S597d
VAR	S54c

When the lexical analyzer cannot recognize a sequence of characters, it calls utility function `unrecognized`. If the sequence is empty, it means there's no token. If anything else happens, an error has occurred.

S600a. ⟨*lexical utility functions for μProlog* S599d⟩+≡ (S598b) ◁S599d S600b▷

```
unrecognized : char list error -> ('a error * 'a error stream) option
```

```
fun unrecognized (ERROR _) = raise InternalError "this can't happen"
  | unrecognized (OK cs) =
      case cs
        of []        => NONE
         | #";" :: _ => raise InternalError "this can't happen"
         | _ =>
             SOME (ERROR ("invalid initial character in `" ^ implode cs ^ "'"), EOS)
```

When a lexical analyzer runs out of characters on a line, it calls `nextline` to compute the location of the next line.

S600b. ⟨*lexical utility functions for μProlog* S599d⟩+≡ (S598b) ◁S600a

```
nextline : srcloc -> srcloc
```

```
fun nextline (file, line) = (file, line+1)
```

μProlog must be aware of the end of an input line. Lexical analyzers `char` and `eol` recognize a character and the end-of-line marker, respectively.

S600c. ⟨*lexical analyzers for for μProlog* S600c⟩≡ (S598b) S600d▷

```
type 'a prolog_lexer
char : char prolog_lexer
eol  : unit prolog_lexer
```

```
type 'a prolog_lexer =
  (char eol_marked, 'a) xformer
fun char chars =
  case streamGet chars
    of SOME (INLINE c, chars) => SOME (OK c, chars)
     | _ => NONE
fun eol chars =
  case streamGet chars
    of SOME (EOL _, chars) => SOME (OK (), chars)
     | _ => NONE
```

Function `manySat` provides a general tool for sequences of characters. Lexers `whitespace` and `intChars` handle two common cases.

S600d. ⟨*lexical analyzers for for μProlog* S600c⟩+≡ (S598b) ◁S600c S600e▷

```
manySat    : (char -> bool) -> char list prolog_lexer
whitespace : char list prolog_lexer
intChars   : char list prolog_lexer
```

```
fun manySat p =
  many (sat p char)

val whitespace =
  manySat isSpace
val intChars =
  (curry op :: <$> eqx #"-" char <|> pure id) <*> many1 (sat Char.isDigit char)
```

An ordinary token is an underscore, delimiter, integer literal, symbolic name, or alphanumeric name. Uppercase and lowercase names produce different tokens.

S600e. ⟨*lexical analyzers for for μProlog* S600c⟩+≡ (S598b) ◁S600d S601a▷

```
ordinaryToken : token prolog_lexer
```

```
val ordinaryToken =
      underscore            <$> eqx #"_" char <*>! manySat isIdent
  <|> int                   <$> intChars      <*>! manySat isIdent
  <|> (RESERVED o str)      <$> sat isDelim char
  <|> (symbolic o implode)  <$> many1 (sat isSymbol char)
  <|> curry (lower o implode o op ::) <$> sat Char.isLower char <*> manySat isIdent
  <|> curry (UPPER o implode o op ::) <$> sat Char.isUpper char <*> manySat isIdent
  <|> unrecognized o fst o valOf o many char
```

I define two main lexical analyzers that keep track of source locations: `tokenAt` produces tokens, and `skipComment` skips comments. They are mutually recursive, and in order to delay the recursive calls until a stream is supplied, each definition has an explicit cs argument, which contains a stream of inline characters.

S601a. ⟨*lexical analyzers for for µProlog* S600c⟩+≡ (S598b) ◁S600e

```
                  tokenAt     : srcloc -> token located prolog_lexer
                  skipComment : srcloc -> srcloc -> token located prolog_lexer
local
  fun the c = eqx c char
in
  fun tokenAt loc cs =  (* eta-expanded to avoid infinite regress *)
    (whitespace *> (   the #"/" *> the #"*" *> skipComment loc loc
                   <|> the #";" *> many char *> eol *> tokenAt (nextline loc)
                   <|>                          eol *> tokenAt (nextline loc)
                   <|> (loc, EOF) <$ eos
                   <|> pair loc <$> ordinaryToken
                   )) cs
  and skipComment start loc cs =
    (   the #"*" *> the #"/" *> tokenAt loc
    <|> char *> skipComment start loc
    <|> eol  *> skipComment start (nextline loc)
    <|> id <$>! pure (ERROR ("end of file looking for */ to close comment in " ^
                             srclocString start))
    ) cs
end
```

V.7 Parsing

Not much parsing code is shared with other interpreters.

S601b. ⟨*lexical analysis and parsing for µProlog, providing* xdefsInMode S598b⟩+≡ (S593a) ◁S598b

⟨*parsers and streams for µProlog* S601c⟩
```
val xdefstream = xdefsInMode RMODE
```
⟨*shared definitions of* filexdefs *and* stringsxdefs S233a⟩

The one thing that's interesting about µProlog's parser is that a µProlog interpreter has two modes: rule mode and query mode. Tracking modes adds complexity to the parser.

V.7.1 Utilities for parsing µProlog

As always, the first combinators to be defined are those that parse single tokens.

S601c. ⟨*parsers and streams for µProlog* S601c⟩≡ (S601b) S602a▷

```
                                                    symbol : string parser
                                                    upper  : string parser
                                                    lower  : string parser
                                                    int    : int    parser
type 'a parser = (token, 'a) polyparser
val symbol = asAscii ((fn SYMBOLIC  s => SOME s | _ => NONE) <$>? token)
val upper  = asAscii ((fn UPPER     s => SOME s | _ => NONE) <$>? token)
val lower  = asAscii ((fn LOWER     s => SOME s | _ => NONE) <$>? token)
val int    =          (fn INT_TOKEN n => SOME n | _ => NONE) <$>? token
fun reserved s = eqx s ((fn RESERVED s => SOME s | _ => NONE) <$>? token)
```

Unlike more sophisticated languages, μProlog does not have a system of operator precedence and associativity. That means an input like `3 + X + Y` is not a well-formed term. To ensure that an input like `3 + X` is not followed by another symbol, the parser uses `notSymbol`.

S602a. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S601c S602b▷

```
notSymbol : unit parser
```

```
val notSymbol =
  symbol <!> "arithmetic expressions must be parenthesized" <|>
  pure ()
```

Like ML, μProlog provides syntactic sugar for lists. To turn this sugar into list terms, the parser uses term `nilt` and function `cons`.

S602b. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S602a S602c▷

```
val nilt = APPLY ("nil", [])
fun cons (x, xs) = APPLY ("cons", [x, xs])
```

```
nilt : term
cons : term * term -> term
```

Lexical functions `upper`, `symbol`, and `lower` are renamed to show their meaning as μProlog tokens. And utility function `commas` builds a parser that parses things separated by commas.

S602c. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S602b S602d▷

```
val variable        = upper
val binaryPredicate = symbol
val functr          = lower
fun commas p =
  curry op :: <$> p <*> many (reserved "," *> p)
```

```
variable        : string parser
binaryPredicate : string parser
functr          : string parser
commas : 'a parser -> 'a list parser
```

I spell "functor" without the "o" because in Standard ML, `functor` is a reserved word.

Parser `wrap` L R p is used to parse whatever p parses, but wrapped in tokens on the left and right. Tokens L and R will be either round brackets or square brackets.

S602d. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S602c S603a▷

```
wrap : string -> string -> 'a parser -> 'a parser
```

```
fun closing bracket = reserved bracket <?> bracket
fun wrap left right p = reserved left *> p <* closing right
```

V.7.2 Parsing terms, atoms, and goals

The elements above can now be combined into a parser for μProlog. The grammar is based on the grammar from Figure D.2 on page S52, except that atoms are parsed using named functions, and I use some specialized tricks to organize the grammar. Concrete syntax is not for the faint of heart.

S603a. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S602d S603b▷

```
term   : term parser
atom   : term parser
commas : 'a parser -> 'a list parser
```

```
local
  fun consElems terms tail = foldr cons tail terms
  fun applyIs a t = APPLY ("is", [a, t])
  fun applyBinary x operator y = APPLY (operator, [x, y])
  fun maybeClause t NONE = t
    | maybeClause t (SOME ts) = APPLY (":-", t :: ts)
in
  fun term tokens =
    (   applyIs <$> atom <* reserved "is" <*> (term <?> "term")
    <|> applyBinary <$> atom <*> binaryPredicate <*> (atom <?> "atom") <* notSymbol
    <|> atom
    )
    tokens
  and atom tokens =
    (   curry APPLY <$> functr <*> (wrap "(" ")" (commas (term <?> "term"))
                                   <|> pure []
                                   )
    <|> VAR     <$> variable
    <|> LITERAL <$> int
    <|> wrap "(" ")" (maybeClause <$> term <*> optional (reserved ":-" *> commas term))
    <|> wrap "[" "]"
            (consElems <$> commas term <*> ( reserved "|" *> (term <?> "list ele
                                           <|> pure nilt
                                            )
           <|> pure nilt
            )
    )
    tokens
end
```

Terms and goals share the same concrete syntax but different abstract syntax. Every goal can be interpreted as a term, but not every term can be interpreted as a goal.

S603b. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S603a S604a▷

```
asGoal : srcloc -> term -> goal error
goal   : goal parser
```

```
fun asGoal _   (APPLY g) = OK g
  | asGoal loc (VAR v)   =
      synerrorAt ("Variable " ^ v ^ " cannot be a predicate") loc
  | asGoal loc (LITERAL n) =
      synerrorAt ("Integer " ^ intString n ^ " cannot be a predicate") loc

val goal = asGoal <$> srcloc <*>! term
```

V.7.3 Recognizing concrete syntax using modes

I put together the μProlog parser in three layers. The bottom layer is the concrete syntax itself. For a moment let's ignore the *meaning* of μProlog's syntax and look only at what can appear. At top level, we might see

- A string in brackets
- A clause containing a `:-` symbol
- A list of one or more goals separated by commas
- A unit test

The meanings of some of these things can be depend on which mode the interpreter is in. So each one is parsed initially into a value of type `concrete`, which represents the concrete syntax while taking no position on what the syntax means. This representation can be interpreted as abstract syntax later, once the mode is known.

S604a. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁ S603b S604b ▷

`type concrete`

```
datatype concrete
  = BRACKET of string
  | CLAUSE  of goal * goal list option
  | GOALS   of goal list
  | CTEST   of unit_test
```

One form of `concrete` is a `unit_test`. All unit tests are parsed similarly, but parsing `check-satisfied` is a bit tricky: the concrete syntax provides a list of goals, which must be split into "real" goals `gs'` and "substitution" goals `rest`. A "substitution" goal is an application of the = functor.

S604b. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁ S604a S604c ▷

`checkSatisfied : goal list -> unit_test error`

```
fun checkSatisfied goals =
  let fun split (gs', []) = OK (CHECK_SATISFIED (reverse gs', []))
        | split (gs', rest as ("=", _) :: _) =
             validate ([], rest) >>=+
             (fn subst => CHECK_SATISFIED (reverse gs', subst))
        | split (gs', g :: gs) = split (g :: gs', gs)
      and validate (theta', ("=", [VAR x, t]) :: gs) =
            validate ((x, t) :: theta', gs)
        | validate (theta', ("=", [t1, t2]) :: gs) =
            ERROR ("in check_satisfied, " ^ termString t1 ^ " is set to " ^
                   termString t2 ^ ", but " ^ termString t1 ^ " is not a variable")
        | validate (theta', g :: gs) =
            ERROR ("in check_satisfied, expected a substitution but got " ^
                   goalString g)
        | validate (theta', []) = OK (reverse theta')
  in  split ([] , goals)
  end
```

The three unit tests are recognized and treated specially.

S604c. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁ S604b S605a ▷

`unit_test : unit_test parser`

```
val unit_test =
     reserved "check_satisfiable" *>
        (wrap "(" ")" (CHECK_SATISFIABLE <$> commas goal)
        <?> "check_satisfiable(goal, ...)")
 <|> reserved "check_unsatisfiable" *>
        (wrap "(" ")" (CHECK_UNSATISFIABLE <$> commas goal)
        <?> "check_unsatisfiable(goal, ...)")
 <|> reserved "check_satisfied" *>
        (wrap "(" ")" (checkSatisfied <$>! commas goal)
         <?> "check_satisfied(goal, ... [, X1 = t1, ...])")
```

Given the unit_test parser, the remaining forms of concrete are easy to recognize.

S605a. ⟨*parsers and streams for µProlog* S601c⟩+≡ (S601b) ◁S604c S605b▷

```
concrete : concrete parser
```

```
val notClosing =
  sat (fn RESERVED "]" => false | _ => true) token
val concrete =
     (BRACKET o concat o map tokenString) <$> wrap "[" "]" (many notClosing)
 <|> CTEST <$> unit_test
 <|> curry CLAUSE <$> goal <*> reserved ":-" *> (SOME <$> commas goal)
 <|> GOALS <$> commas goal
```

In most contexts, the parser knows what a concrete value is supposed to mean, but there's one case in which it doesn't: a phrase like "color(yellow)." could be either a clause *or* a query. To know which is meant, the parser has to know the *mode*. In other words, the mode distinguishes CLAUSE(g, NONE) from GOALS [g]. A parser may be in either query mode or rule (clause) mode. Each mode has its own prompt.

S605b. ⟨*parsers and streams for µProlog* S601c⟩+≡ (S601b) ◁S605a S605c▷

```
type mode
mprompt : mode -> string
```

```
datatype mode = QMODE | RMODE
fun mprompt RMODE = "-> "
  | mprompt QMODE = "?- "
```

The concrete syntax normally means a clause or query, which is denoted by the syntactic nonterminal symbol *clause-or-query* and represented by an ML value of type cq (see chunk S55a in Appendix D). But particular concrete syntax, such as "[rule]." or "[query].," can be an instruction to change to a new mode. The middle layer of µProlog's parser produces a value of type xdef_or_mode, which is defined as follows:

S605c. ⟨*parsers and streams for µProlog* S601c⟩+≡ (S601b) ◁S605b S606a▷

```
type xdef_or_mode
```

```
datatype xdef_or_mode
  = XDEF of xdef
  | NEW_MODE of mode
```

The next level of μProlog's parser interprets a `concrete` value according to the mode. `BRACKET` values and unit tests are interpreted in the same way regardless of mode, but clauses and especially `GOALS` are interpreted differently in rule mode and in query mode.

S606a. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S605c S606b▷

`interpretConcrete : mode -> concrete -> xdef_or_mode error`

```
fun interpretConcrete mode =
  let val (newMode, cq, xdef) = (OK o NEW_MODE, OK o XDEF o DEF, OK o XDEF)
  in  fn c =>
        case (mode, c)
          of (_, BRACKET "rule")     => newMode RMODE
           | (_, BRACKET "fact")     => newMode RMODE
           | (_, BRACKET "user")     => newMode RMODE
           | (_, BRACKET "clause")   => newMode RMODE
           | (_, BRACKET "query")    => newMode QMODE
           | (_, BRACKET s)          => xdef (USE s)
           | (_, CTEST t)            => xdef (TEST t)
           | (RMODE, CLAUSE (g, ps)) => cq (ADD_CLAUSE (g :- getOpt (ps, [])))
           | (RMODE, GOALS [g])      => cq (ADD_CLAUSE (g :- []))
           | (RMODE, GOALS _ ) =>
               ERROR ("You cannot enter a query in clause mode; " ^
                      "to change modes, type `[query].'")
           | (QMODE, GOALS gs)           => cq (QUERY gs)
           | (QMODE, CLAUSE (g, NONE))   => cq (QUERY [g])
           | (QMODE, CLAUSE (_, SOME _)) =>
               ERROR ("You cannot enter a new clause in query mode; " ^
                      "to change modes, type `[rule].'")
  end
```

Parser `xdef_or_mode` m parses a `concrete` according to mode m. If it sees something it doesn't recognize, it emits an error message and skips ahead until it sees a dot or the end of the input. Importantly, this parser never fails: it always returns either a `xdef_or_mode` value or an error message.

S606b. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S606a S607a▷

`xdef_or_mode : mode -> xdef_or_mode parser`

```
val skippable =
  (fn SYMBOLIC "." => NONE | EOF => NONE | t => SOME t) <$>? token

fun badConcrete (loc, skipped) last =
  ERROR (srclocString loc ^ ": expected clause or query; skipping" ^
         concat (map (fn t => " " ^ tokenString t) (skipped @ last)))

fun xdef_or_mode mode = interpretConcrete mode <$>!
  (   concrete <* reserved "."
  <|> badConcrete <$> @@ (many  skippable) <*>! ([RESERVED "."] <$ reserved ".")
  <|> badConcrete <$> @@ (many1 skippable) <*>! pure []  (* skip to EOF *)
  )
```

V.7.4 Reading clauses and queries while tracking locations and modes

To produce a stream of definitions, every other interpreter in this book uses the function `interactiveParsedStream` from page S269. μProlog can't: function `interactiveParsedStream` doesn't keep track of modes. As a replacement, I define a somewhat more complex function, `xdefsInMode`, below. At the core of `xdefsInMode` is function `getXdef`.

S607a. ⟨*parsers and streams for μProlog* S601c⟩+≡ (S601b) ◁S606b

```
xdefsInMode : mode -> string * line stream * prompts -> xdef stream
type read_state = string * mode * token located eol_marked stream
getXdef : read_state -> (xdef * read_state) option
```

```
fun xdefsInMode initialMode (name, lines, prompts) =
  let val { ps1, ps2 } = prompts
      val thePrompt = ref (if ps1 = "" then "" else mprompt initialMode)
      val setPrompt = if ps1 = "" then (fn _ => ()) else (fn s => thePrompt := s

      type read_state = string * mode * token located eol_marked stream
      ⟨utility functions for xdefsInMode S607b⟩

      val lines = preStream (fn () => print (!thePrompt), echoTagStream lines)

      val chars =
        streamConcatMap
        (fn (loc, s) => streamOfList (map INLINE (explode s) @ [EOL (snd loc)]))
        (locatedStream (name, lines))

      fun getLocatedToken (loc, chars) =
        (case tokenAt loc chars
           of SOME (OK (loc, t), chars) => SOME (OK (loc, t), (loc, chars))
            | SOME (ERROR msg,   chars) => SOME (ERROR msg,   (loc, chars))
            | NONE => NONE
        ) before setPrompt ps2

      val tokens =
        stripAndReportErrors (streamOfUnfold getLocatedToken ((name, 1), chars))

  in  streamOfUnfold getXdef (!thePrompt, initialMode, streamMap INLINE tokens)
  end
```

Using INLINE may look strange, but many of the utility functions from Appendix I expect a stream of tokens tagged with INLINE. I want to use those functions, and even though the INLINE tag isn't useful for parsing μProlog, it is easy enough to ignore INLINE. Much easier than rewriting big chunks of Appendix I.

Function getXdef, which is defined below, uses startsWithEOF to check if the input stream has no more tokens.

S607b. ⟨*utility functions for* xdefsInMode S607b⟩≡ (S607a) S607c▷

```
startsWithEOF : token located eol_marked stream -> bool
```

```
fun startsWithEOF tokens =
  case streamGet tokens
    of SOME (INLINE (_, EOF), _) => true
     | _ => false
```

If getXdef detects an error, it skips tokens in the input up to and including the next dot.

S607c. ⟨*utility functions for* xdefsInMode S607b⟩+≡ (S607a) ◁S607b S608a▷

```
skipPastDot : token located eol_marked stream ->
                  token located eol_marked stream
```

```
fun skipPastDot tokens =
  case streamGet tokens
    of SOME (INLINE (_, RESERVED "."), tokens) => tokens
     | SOME (INLINE (_, EOF), tokens) => tokens
     | SOME (_, tokens) => skipPastDot tokens
     | NONE => tokens
```

And now the definition of `getXdef`. It tracks the prompt, the mode, and the remaining unread tokens, which together form the `read_state`. It also, when called, sets the prompt.

S608a. ⟨*utility functions for* `xdefsInMode` S607b⟩+≡ (S607a) ◁S607c

`getXdef : read_state -> (xdef * read_state) option`

```
fun getXdef (ps1, mode, tokens) =
  ( setPrompt ps1
  ; if startsWithEOF tokens then
      NONE
    else
      case xdef_or_mode mode tokens
        of SOME (OK (XDEF d),        tokens) => SOME (d, (ps1, mode, tokens))
         | SOME (OK (NEW_MODE mode), tokens) => getXdef (mprompt mode, mode, tokens)
         | SOME (ERROR msg,          tokens) =>
                                         ( eprintln ("syntax error: " ^ msg)
                                         ; getXdef (ps1, mode, skipPastDot tokens)
                                         )
         | NONE => ⟨fail epically with a diagnostic about tokens S608b⟩
  )
```

Parser `xdef_or_mode` is always supposed to return something. If it doesn't, the interpreter dumps all the tokens and fails with an internal error.

S608b. ⟨*fail epically with a diagnostic about* `tokens` S608b⟩≡ (S608a)

```
let val tokensStrings =
      map (fn t => " " ^ tokenString t) o valOf o peek (many token)
    val _ = app print (tokensStrings tokens)
in  raise InternalError "cq parser failed"
end
```

V.8 Altered infrastructure

A couple of the standard functions used in other interpreters have been tweaked to work better with μProlog and with implementations of real Prolog.

V.8.1 Altered `useFile`

To make μProlog more compatible with other implementations of Prolog, I patch the `useFile` function defined in Chapter 5. If the original `useFile` (here called `try`) fails with an I/O error, the patched version tries adding "`.P`" to the name; this is the convention used by XSB Prolog. If adding `.P` fails, it also tries "`.pl`"; this is the convention used by GNU Prolog and SWI Prolog.

S608c. ⟨*definition of* `useFile`*, to read from a file* S608c⟩≡ (S238b)

```
val try = useFile
fun useFile filename =
  try filename          handle IO.Io _ =>
  try (filename ^ ".P") handle IO.Io _ =>
  try (filename ^ ".pl")
```

V.8.2 Altered command-line processing

*μ*Prolog's command-line processor differs from our other interpreters, because it has to deal with modes. When prompting, it starts in query mode; when not prompting, it starts in rule mode. And because I'm pressed for time, it doesn't have all the nice option processing of the other command-line processors.

S609a. ⟨*function* `runAs` *for μProlog* S609a⟩≡ (S593a)

`runAs : interactivity -> unit`

```
fun runAs interactivity =
  let val _ = setup_error_format interactivity
      val (prompts, prologMode) =
        if prompts interactivity then (stdPrompts, QMODE) else (noPrompts, RMODE)
      val xdefs =
        xdefsInMode prologMode ("standard input", filelines TextIO.stdIn, prompts)
  in  ignore (readEvalPrintWith eprintln (xdefs, emptyDatabase, interactivity))
  end
```

The `-q` option is as in other interpreters, and the `-trace` option turns on tracing.

S609b. ⟨*code that looks at μProlog's command-line arguments and calls* `runAs` S609b⟩≡ (S593a)

```
fun runmain ["-q"]          = runAs (NOT_PROMPTING, ECHOING)
  | runmain []              = runAs (PROMPTING,     ECHOING)
  | runmain ("-trace" :: t) = (tracer := app eprint; runmain t)
  | runmain _  =
      TextIO.output (TextIO.stdErr,
                     "Usage: " ^ CommandLine.name() ^ " [trace] [-q]\n")
val _ = runmain (CommandLine.arguments())
```

If you attempt any of the exercises, your code can call `trace`, which will print things if and only if the `-trace` option is given to the interpreter.

S609c. ⟨*support for tracing μProlog computation* S609c⟩≡ (S593a)

`trace : string list -> unit`

```
val tracer = ref (app print)
val _ = tracer := (fn _ => ())
fun trace l = !tracer l
```

V

Supporting code for μProlog

Part VII. Code index

Code index