Programming Languages

Build, Prove, and Compare (The Supplement)

Norman Ramsey
Tufts University
Medford, Mass

For COMP 105, Tufts University, Fall 2020
August, 2020

Preface to the supplement

This volume is the Supplement to *Programming Languages: Build, Prove, and Compare*.

Contents

V. Supplemental topics

EBNF

A

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. In this book, we use extended BNF, or simply EBNF, which makes it easier to specify optional and repeated items (Wirth 1977).

An EBNF grammar consists of 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 non-terminal symbol represents all the phrases in a syntactic category. Thus, *toplevel* represents all legal top-level inputs, *exp* all legal expressions, and *name* all legal names. Literal text, on the other hand, represents characters that appear *as is* in syntactic phrases.

Consider the rule for *toplevel* in Impcore:

toplevel ::= *exp*
    | `(use` *file-name*`)`
    | `(val` *variable-name exp*`)`
    | `(define` *function-name* `(`*formals*`)` *exp*`)`

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

- A legal *exp*
- A left parenthesis followed by the word `use`, then a file name, and then a right parenthesis
- A left parenthesis followed by the word `val`, then a variable name, then a legal *exp*, and then a right parenthesis
- 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 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.

For another example, the phrase

```
(set x 10)
```

is a *toplevel* input by the following reasoning:

- An *input* can be an *expression*.
- An *expression* can be a left parenthesis and the word `set`, followed by a *variable* and an *expression*, followed by a right parenthesis.
- A *variable* is a *name*, and a *name* is a sequence of characters which may be the sequence "x" (we appeal to the English description of this category).
- An *expression* can be a *value*, a *value* is an *integer*, and `10` is an *integer*.

The explanation above is not the whole story. In addition to a nonterminal symbol or literal text, a sequence may contain a collection of alternatives in brackets. EBNF offers three kinds of brackets:

- 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.

In each case, alternatives within brackets are separated by a vertical bar ($|$).

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

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

Similarly, the EBNF phrase "`(`*function-name* $\left\{\textit{exp}\right\}$`)`" 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).

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. So many languages provide arbitrary-precision arithmetic on integers or rational numbers that you don't even need to know how the tricks are done. This supplemental chapter, 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 S16. There are just 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, we compute quotient and remainder together. (This is true even in hardware.) Storing quotient and remainder is the purpose of record type `QR.pair`.

S16a. ⟨*nat.mcl* S16a⟩≡

```
(module-type NATURAL
  (exports [abstype t]
           [of-int : (int -> t)]   ; creator
           [+ : (t t -> t)]        ; producer
           [- : (t t -> t)]        ; producer
           [* : (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
```

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, we divide a natural number only by a (positive) machine integer. This "short division" is implemented by function `sdiv`.

A natural number can be represented easily and efficiently as a sequence of *digits* in a given *base*. The algorithms for basic 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 $0 \leq x_i < b$. 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 representation of a sequence of digits. I discuss two representations:

- We 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 a digit and X' is a natural number. (It is possible to begin with x_n instead of x_0, but the so-called "little-endian" representation, with the least-significant digit on the left, simplifies all the computations.) A suitable representation might use an algebraic data type (Chapters 8 and 9):

  S16b. ⟨*representation of natural numbers as a list of digits* S16b⟩≡

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

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

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 $E = m \cdot c^2$.

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

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

- Alternatively, we 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 $0 \leq x_i < b$.

Here are the design tradeoffs: 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. The rest of this section shows algorithms for both representations.

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.

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

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 ($X \operatorname{div} b, Y \operatorname{div} b$)
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 ($Q \operatorname{div} b$)
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, $0 \le r_i < d$

Table B.2: Metavariables used to describe multiprecision arithmetic

$Y = y_0 + Y' \cdot b$. 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:

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

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.

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 $sbb(X, Y, c) = X - Y - c$:

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

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 use these equations:

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

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 up 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.

To multiply two natural numbers represented as arrays, we compute

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

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 `print`: to convert a large integer to a sequence of decimal digits, we divide it 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:

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

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,

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

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:

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

In the example on page S21,

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

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.

S23. ⟨*Molecule's predefined module types* S23⟩≡

```
(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

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 $\mathrm{mod}\ b$ and $\mathrm{div}\ b$ 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 10.7, 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:

$$+N + +M = +(N + M) \qquad +N < +M = N < M$$
$$+N + -M = +(N - M), \text{ when } N \geq M \qquad +N < -M = \texttt{\#f}$$
$$+N + -M = -(M - N), \text{ when } N < M \qquad \texttt{negated}(+N) = -N$$
$$+N + 0 = +N \qquad \texttt{negated}(0) = 0$$

The implementation of these laws depends on the programming language. If we're using abstract data types in Molecule, our code can inspect the representations of two integers at once, and the signed-integer operations can be implemented by pattern matching on pairs. If we're using objects in μSmalltalk, our code will have to identify some representations using double dispatch (Section 10.7.3).

Chapter contents

C

Extensions to algebraic data types

As I write this chapter, 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. It's possible to get algebraic data types to do more, and in this section 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 the Supplement. 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. Sadly, 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 S29 below, which you can compare with the opening example of Chapter 10. But before we look the example, we had better see 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:

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

Extensions to algebraic data types

C

S28

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.

S28a. ⟨*existential transcript* S28a⟩≡ S28b ▷

```
-> (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:

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

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

S28c. ⟨*existential transcript* S28a⟩+≡ ◁ S28b S28d ▷

```
-> (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:

S28d. ⟨*existential transcript* S28a⟩+≡ ◁ S28c S28e ▷

```
-> (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]

S28e. ⟨*existential transcript* S28a⟩+≡ ◁ S28d S28f ▷

```
-> (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:

S28f. ⟨*existential transcript* S28a⟩+≡ ◁ S28e S29a ▷

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

S29a. ⟨*existential transcript* S28a⟩+≡ ◁S28f S29b▷

```
-> (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 21" is an "escaping skolem type." The *skolem type* (page S33, 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.

S29b. ⟨*existential transcript* S28a⟩+≡ ◁S29a S29c▷

```
-> (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:

S29c. ⟨*existential transcript* S28a⟩+≡ ◁S29b S29d▷

```
-> (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.

S29d. ⟨*existential transcript* S28a⟩+≡ ◁S29c S29e▷

```
-> (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.

S29e. ⟨*existential transcript* S28a⟩+≡ ◁S29d S29f▷

I start by scaling points and integers.

S29f. ⟨*existential transcript* S28a⟩+≡ ◁S29e S30a▷

```
-> (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.

S30a. ⟨*existential transcript* S28a⟩+≡ ◁S29f S30b▷

```
                         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))]))
```

Extensions to algebraic data types

S30

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:

S30b. ⟨*existential transcript* S28a⟩+≡ ◁S30a S30c▷

```
        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).

S30c. ⟨*existential transcript* S28a⟩+≡ ◁S30b S30d▷

```
-> (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:

S30d. ⟨*existential transcript* S28a⟩+≡ ◁S30c S31a▷

```
-> (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:

S31a. ⟨*existential transcript* S28a⟩+≡ ◁S30d S31b▷

```
-> (define translate-shape (vector s)      translate-shape : (pt shape -> shape)
     (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:

S31b. ⟨*existential transcript* S28a⟩+≡ ◁S31a S31c▷

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

S31c. ⟨*existential transcript* S28a⟩+≡ ◁S31b S31d▷

```
-> (implicit-data circle [C of pt int])    circle : (pt int -> shape)
                             ; (C center radius)
-> (use postscript.uml) ;; load PostScript drawing library from Supplement
-> (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.

S31d. ⟨*existential transcript* S28a⟩+≡ ◁S31c S31e▷

```
-> (val disk                               disk : (pt int -> shape)
     (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.

S31e. ⟨*existential transcript* S28a⟩+≡ ◁S31d S32a▷

```
                                           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.

S32a. ⟨*existential transcript* S28a⟩+≡ ◁S31e S32b▷

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

S32b. ⟨*existential transcript* S28a⟩+≡ ◁S32a S32c▷

```
-> (val target                                                    target : shape
      (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:

S32c. ⟨*existential transcript* S28a⟩+≡ ◁S32b

```
-> (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 it:

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 S28). 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 arose most probably 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:

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

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.

Here's the rule:

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

Function ftc 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 $\tilde{\mu}_{17}$. It is absolutely crucial that $\tilde{\mu}_{17}$ not *escape* the case expression. That's because the equivalence $\tilde{\mu}_{17} \equiv \tilde{\mu}_{17}$ is sound only for duration of a single evaluation. We must prevent all the following means of escape:

- A skolem type appears in the type of the result.
- A skolem type appears in the type of the scrutinee.
- A skolem type appears in a constraint in such a way that it wants to be substituted for a type variable that appears free in the environment.

So 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.

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

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.

S35a. ⟨*transcript* S35a⟩≡ S35b ▷

```
-> (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:

S35b. ⟨*transcript* S35a⟩+≡ ◁ S35a S35c ▷

```
-> (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 when we 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`, I still have 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:

  S35c. ⟨*transcript* S35a⟩+≡ ◁ S35b

  ```
  -> (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 on page S39. 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]

S36a. ⟨*Haskell definitions for GADT example* S36a⟩≡ S36b ▷

```
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.

S36b. ⟨*Haskell definitions for GADT example* S36a⟩+≡ ◁ S36a S36c ▷

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

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

S36c. ⟨*Haskell definitions for GADT example* S36a⟩+≡ ◁ S36b S37b ▷

```
int  :: (Int  -> (TypedExp Int))
bool :: (Bool -> (TypedExp Bool))
plus :: ((TypedExp Int) -> (TypedExp Int) -> (TypedExp Int))
less :: ((TypedExp Int) -> (TypedExp Int) -> (TypedExp Bool))
```

[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.

```
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.

S37a. ⟨*GHCI transcript* S37a⟩≡ S38b ▷

```
*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:

S37b. ⟨*Haskell definitions for GADT example* S36a⟩+≡ ◁ S36c S37c ▷

```
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:

S37c. ⟨*Haskell definitions for GADT example* S36a⟩+≡ ◁ S37b S38a ▷

```
data TExp :: * -> * where
```

```
TLit  :: forall a . (a -> (TExp a)) -- XXX fix me
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)) -- XXX fix me
```

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:

S38a. ⟨*Haskell definitions for GADT example* S36a⟩+≡ ◁ S37c

```
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:

S38b. ⟨*GHCI transcript* S37a⟩+≡ ◁ S37a

```
*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. As of early 2015, the Glasgow Haskell Compiler uses 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.

More GADTs

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 on page S206.

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 will 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:

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

You will reuse the `txTyScheme` function from chunk S427b, 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.

Chapter contents

C.4.1 Bonus features

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*

The problem that led to the creation of Prolog was the problem of creating machine intelligence. Alan Turing's famous test deems a machine 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 such 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. Judgments, syntactic proofs, inference rules, and valid derivations—in other words, all of our operational semantics and type theory—are from mathematical logic. 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`, `stephen_hawking`, `z`, `table`, and `smallmouth`. 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.

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.

[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.

- 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:

S48. ⟨*transcript* S48⟩≡ S49a ▷

```
?- 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 S129 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:

S49a. ⟨*transcript* S48⟩+≡ ◁S48 S49b▷
```
?- [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 on page S109). In language of inference rules, the rule "every psychiatrist is 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:

S49b. ⟨*transcript* S48⟩+≡ ◁S49a S50a▷
```
-> 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:

S50a. ⟨*transcript* S48⟩+≡ ◁S49b S50b▷

```
-> 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?

S50b. ⟨*transcript* S48⟩+≡ ◁S50a S50c▷

```
-> [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?"

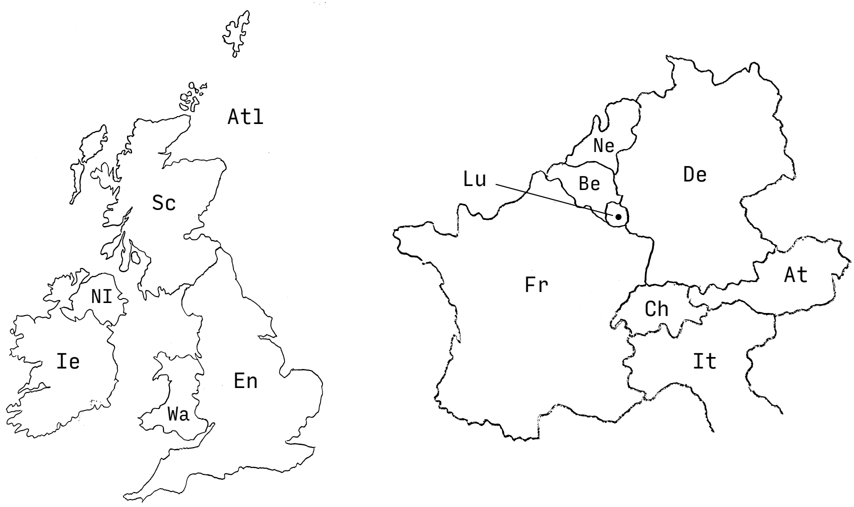
S50c. ⟨*transcript* S48⟩+≡ ◁S50b S50d▷

```
?- 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."

S50d. ⟨*transcript* S48⟩+≡ ◁S50c S51▷

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



(a) The British Isles　　(b) Part of Western Europe

Figure D.1: Maps

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:

S51. ⟨*transcript* S48⟩+≡ ◁S50d S52a▷

```
-> [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).
```

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.1 (a) 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:

S52a. ⟨*transcript* S48⟩+≡ ◁S51 S52b▷

```
-> 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.1 (a) 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:

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

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

S52b. ⟨*transcript* S48⟩+≡ ◁S52a S52c▷

```
-> 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):

S52c. ⟨*transcript* S48⟩+≡ ◁S52b S53a▷

```
-> [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*. Here's how we ask for a coloring of map (a):

S53a. ⟨*transcript* S48⟩+≡ ◁ S52c S53b ▷

```
?- 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:

S53b. ⟨*transcript* S48⟩+≡ ◁ S53a S53c ▷

```
?- 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:

S53c. ⟨*transcript* S48⟩+≡ ◁ S53b S54 ▷

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

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

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`]` 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:

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

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

S54. ⟨*transcript* S48⟩+≡ ◁S53c S55a▷

```
?- [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.

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

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

S55a. ⟨*transcript* S48⟩+≡ ◁S54 S55b▷

```
-> [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:

S55b. ⟨*transcript* S48⟩+≡ ◁S55a S57a▷

```
?- 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.

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 on page S72.

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.

[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 `=<`.

> 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 $g_1, \ldots, g_n$.
- Test `check_satisfied(`$g_1, \ldots, g_n$`,` X_1 `=` $t_1, \ldots, X_m$ `=` t_m`)` gives both a query and a substitution that is supposed to satisfy it. The test passes if the query is satisfied by the *particular* 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:

S57a. ⟨*transcript* S48⟩+≡ ◁S55b S57b▷

```
?- 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.

S57b. ⟨*transcript* S48⟩+≡ ◁S57a S60a▷

```
?- 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.)

S58a. ⟨*definitions of* `term`, `goal`, *and* `clause` *for μProlog* S58a⟩≡ (S58f) S58b ▷

```
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.

S58b. ⟨*definitions of* `term`, `goal`, *and* `clause` *for μProlog* S58a⟩+≡ (S58f) ◁ S58a S58c ▷

```
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.

S58c. ⟨*definitions of* `term`, `goal`, *and* `clause` *for μProlog* S58a⟩+≡ (S58f) ◁ S58b

```
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.5.3.) 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`.

S58d. ⟨*definitions of* `def` *and* `unit_test` *for μProlog* S58d⟩≡ (S58f) S58e ▷

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

μProlog includes three unit-test forms.

S58e. ⟨*definitions of* `def` *and* `unit_test` *for μProlog* S58d⟩+≡ (S58f) ◁ S58d

```
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.

S58f. ⟨*abstract syntax for μProlog* S58f⟩≡ (S87a)

⟨*definitions of* `term`, `goal`, *and* `clause` *for μProlog* S58a⟩
⟨*definitions of* `def` *and* `unit_test` *for μProlog* S58d⟩
⟨*definition of* `xdef` *(shared)* generated automatically⟩
⟨*definitions of* `termString`, `goalString`, *and* `clauseString` S573c⟩

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 on page S96.) 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:

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

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

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

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."

S60a. ⟨*transcript* S48⟩+≡ ◁ S57b S60b ▷

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

Logically, the question we're 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?"

S60b. ⟨*transcript* S48⟩+≡ ◁ S60a S61a ▷

```
?- 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. Importantly, 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:

$$\theta(\textsc{apply}(f, t_1, \ldots, t_n)) = \textsc{apply}(f, \theta(t_1), \ldots, \theta(t_n))$$
$$\theta(\textsc{literal}(n)) = \textsc{literal}(n)$$

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

Substitutions have the following properties, which you might like to confirm (Exercise 36 on page S117):

- 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 $\theta(g) = \texttt{member(3, [Y,4|ZS])}$.

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.

S61a. ⟨*transcript* S48⟩+≡ ◁S60b S61b▷

```
-> [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.

S61b. ⟨*transcript* S48⟩+≡ ◁S61a S65▷

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

$$\boxed{D \vdash \theta(gs)}$$

EmptyQuery

$$\frac{}{D \vdash I([])}$$

NonemptyQuery

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

Figure D.3: The logical interpretation of Prolog

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 presents a precise, inductive definition of the semantics of Prolog according to the logical interpretation. The 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 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 S64).

  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.

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 S62. 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 on page S97) to control the scope of Prolog's depth-first search.

[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 an 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 = $ `member(3, [3|nil])`
   $g_2 = $ `member(X, [X|XS])`
   $\theta = \{\mathtt{X} \mapsto \mathtt{3},\ \mathtt{XS} \mapsto \mathtt{nil}\}$

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

3. $g_1 = $ `member(3, [4|nil])`
   $g_2 = $ `member(X, [X|XS])`

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

4. $g_1 = $ `length([3|nil], N)`
   $g_2 = $ `member(X, [X|XS])`

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

5. $g_1 = $ `member(X, [X|XS])`
   $g_2 = $ `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

   $\theta(g_1) = $ `member(`t`, [`t`|XS])`
   $\theta(g_2) = $ `member(`t`, [mkTree(`t`, nil, nil)|XS])`

   which cannot be unified: No substitution can make t equal `mkTree(`t`, nil, nil)`, because no matter what you substitute 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 $\{\mathtt{X} \mapsto \mathtt{t}\}$, 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.

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 on page S117). 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 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:

S65. ⟨*transcript* S48⟩+≡ ◁S61b S66▷

```
-> [rule].
-> imokay :- youreokay, hesokay.    /* clause C1 */
-> youreokay :- theyreokay.         /* clause C2 */
-> hesokay.                         /* clause C3 */
-> theyreokay.                      /* clause C4 */
-> [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 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:

S66. ⟨*transcript* S48⟩+≡ ◁S65 S67a▷

```
?- [rule].
-> hesnotokay :- imnotokay.   /* clause C5 */
-> shesokay :- hesnotokay.    /* clause C6 */
-> shesokay :- theyreokay.    /* clause C7 */
-> [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 $C_i = g$ `:-` $H_1, \ldots, H_m$. 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:

S67a. ⟨*transcript* S48⟩+≡ ◁S66 S68▷

```
?- [rule].
-> hesnotokay :- shesokay.   /* clause C8 */
-> hesnotokay :- imokay.     /* clause C9 */
```


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

S67b. ⟨*bad transcript* S67b⟩≡

```
-> [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 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.

[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.

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:

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

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:

S68. ⟨*transcript* S48⟩+≡ ◁ S67a S73 ▷

```
-> [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.

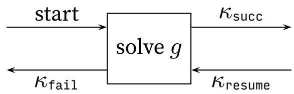
2. Otherwise, find substitution θ_1 that solves goal g_1. If there is no such θ_1, goal g_1 can't be solved.

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 on page 140, 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* $\kappa_{\texttt{succ}}$. 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 $\kappa_{\texttt{succ}}$ 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 $\kappa_{\texttt{resume}}$.

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* $\kappa_{\texttt{fail}}$, which it calls if it can't find a substitution, or if it has to backtrack.

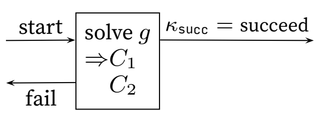
A Byrd box is implemented by a call to function `solveOne` in ⟨*search* **⟦prototype⟧** S84a⟩.■ Byrd boxes are illustrated below by three examples: two based on `member` and one based on map coloring.

The `member` relation has two proof rules:

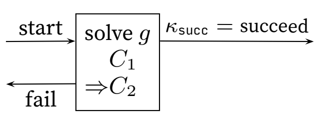
```
member(X, [X|XS]).                    /* C1 */
member(X, [Y|XS]) :- member(X, XS).   /* C2 */
```

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:

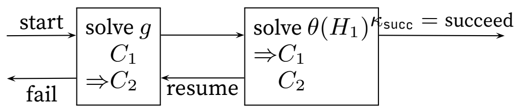
[6]The semantics actually require that we consider all clauses, but these are the only clauses whose heads could possibly unify with query g.

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

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

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 `member(3, [3])`. It creates a new Byrd box. 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.



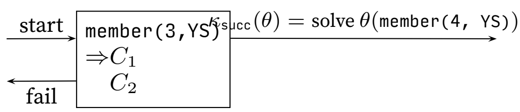
Clause C_1 matches, via $\{\texttt{X2} \mapsto \texttt{3},\ \texttt{XS2} \mapsto \texttt{nil}\}$ (renaming `X` and `XS` in C_1 to `X2` and `XS2`). As C_1 has no subgoals, goal $\theta(H_1)$ is satisfied. Control passes to the success continuation, and query g is also satisfied.

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)`.

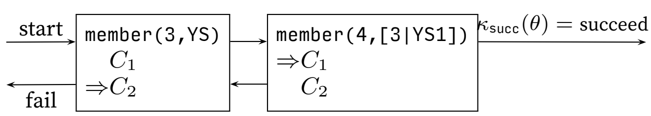


Clause C_1 matches with equality constraint $\texttt{X1} \sim 3 \wedge \texttt{YS} \sim \texttt{[X1|XS1]}$, where `X` and `XS` in C_1 are renamed to `X1` and `XS1`. The constraint is solved by

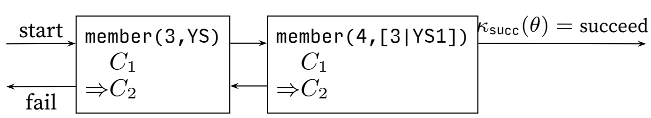
$$\theta_0 = \{\texttt{X1} \mapsto \texttt{3},\ \texttt{YS} \mapsto \texttt{[3|XS1]}\}$$

We pass θ_0 to κ_{succ}.

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.

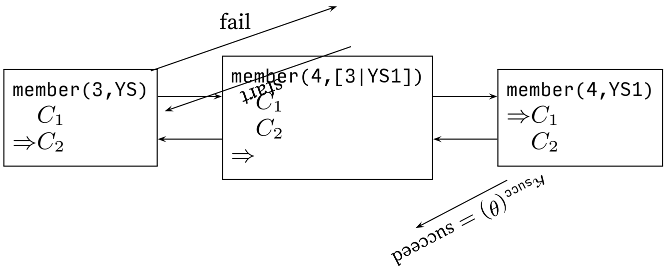
Continuing, the head of C_2 does match the goal: renaming X, Y, and XS in C_2 to X2, Y2, and XS2 produces the equality constraint `member(X2, [Y2|XS2])` $\sim$ `member(4, [3|XS1])`. The constraint is satisfied by

$$\theta'_0 = \{\texttt{X2} \mapsto \texttt{4},\ \texttt{Y2} \mapsto \texttt{3},\ \texttt{XS2} \mapsto \texttt{XS1}\}.$$

3. The search doesn't simply pass θ'_0 to $\kappa_{\texttt{succ}}$; 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

$$\theta_1 = \theta'_1 \circ \theta'_0 = \{\texttt{X3} \mapsto \texttt{4},\ \texttt{YS1} \mapsto \texttt{[4|YS3]},\ \texttt{X2} \mapsto \texttt{4},\ \texttt{Y2} \mapsto \texttt{3},\ \texttt{XS2} \mapsto \texttt{[4|YS3]}\}.$$

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), ...
```

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 $\theta_1 = \{\texttt{Atl} \mapsto \texttt{yellow},\ \texttt{En} \mapsto \texttt{blue}\}$.

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 $\theta_4 = \{\texttt{Sc} \mapsto \texttt{blue}\}$.

[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.

5. Goal $\theta_4(\theta_3(\theta_2(\theta_1($`different(Atl, Wa)`$))))$ = `different(yellow, Wa)` is satisfied by substitution $\theta_5 = \{$`Wa` $\mapsto$ `blue`$\}$.

6. Goal $\theta_5(\theta_4(\theta_3(\theta_2(\theta_1($`different(En, Sc)`$)))))$ = `different(blue, blue)` cannot be satisfied.

7. Backtracking to the previous subgoal, goal `different(yellow, Wa)` is resatisfied, yielding $\theta_5' = \{$`Wa` $\mapsto$ `red`$\}$.

8. Goal $\theta_5'(\theta_4(\theta_3(\theta_2(\theta_1($`different(En, Sc)`$)))))$ is still `different(blue, blue)` and still cannot be satisfied.

9. Backtracking, there are no more substitutions that satisfy `different(yellow, Wa)`. The algorithm backtracks to the previous subgoal, `different(yellow, Sc)`, and it satisfies the subgoal with a new substitution $\theta_4' = \{$`Sc` $\mapsto$ `red`$\}$.

10. Like step 5.

11. Like step 6, but this time the goal is $\theta_5(\theta_4'(\theta_3(\theta_2(\theta_1($`different(En, Sc)`$)))))$ = `different(blue, red)`, and the goal is satisfied. Substitution θ_6 is the identity substitution, which I ignore.

12. Goal $\theta_5(\theta_4'(\theta_3(\theta_2(\theta_1($`different(En, Wa)`$)))))$ = `different(blue, red)`, and the goal is satisfied.

13. Goal $\theta_5(\theta_4'(\theta_3(\theta_2(\theta_1($`different(Ie, NI)`$)))))$ = `different(blue, blue)`, which cannot be satisfied.

More backtracking is needed, but finishing this computation is up to you (Exercise 8 on page S108).

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.

S73. ⟨*transcript* S48⟩+≡ ◁S68 S75b▷

```
-> [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 $\{\texttt{X} \mapsto n,\ \texttt{Y} \mapsto n\}$ 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, here is list membership 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, here is list membership 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 grandfathered as 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.

S75a. ⟨*example queries of* snocced S75a⟩≡ (S75b) S75c ▷

```
?- 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:

S75b. ⟨*transcript* S48⟩+≡ ◁ S73 S75e ▷

```
?- [rule].
-> snocced([], X, [X]).
-> snocced([Y|YS], X, [Y|ZS]) :- snocced(YS, X, ZS).
-> [query].
```
⟨*example queries of* snocced S75a⟩

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:

S75c. ⟨*example queries of* snocced S75a⟩+≡ (S75b) ◁ S75a S75d ▷

```
?- 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 $[3, 4]$?

S75d. ⟨*example queries of* snocced S75a⟩+≡ (S75b) ◁ S75c

```
-> 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:

S75e. ⟨*transcript* S48⟩+≡ ◁ S75b S75f ▷

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

The code can be run in both directions:

S75f. ⟨*transcript* S48⟩+≡ ◁ S75e S76b ▷

```
-> [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

S76a. ⟨*example queries of* appended S76a⟩≡ (S76b) S76d ▷
```
?- 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:

S76b. ⟨*transcript* S48⟩+≡ ◁ S75f S76f ▷
```
?- [rule].
-> appended([], YS, YS).
-> appended([X|XS], YS, [X|ZS]) :- appended(XS, YS, ZS).
-> [query].
⟨example queries of appended S76a⟩
```

The μML function has the same structure:

S76c. ⟨*μML clausal definition of* append S76c⟩≡
```
(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.

S76d. ⟨*example queries of* appended S76a⟩+≡ (S76b) ◁ S76a S76e ▷
```
?- appended([3, 4], [5, 6], ZS).
ZS = [3, 4, 5, 6]
yes
?- appended(XS, YS, [5, 6, 7]).
XS = []
YS = [5, 6, 7]
yes
```

Here is a more sophisticated example in which I split $[5, 6, 7]$ into two nonempty lists. The singleton list $[99]$ cannot be so split:

S76e. ⟨*example queries of* appended S76a⟩+≡ (S76b) ◁ S76d
```
?- [rule].
-> nonempty([X|XS]).
-> [query].
?- 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 of using appended in the backward direction, I use appended to define list membership:

S76f. ⟨*transcript* S48⟩+≡ ◁ S76b S77a ▷
```
?- [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 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:

S77a. ⟨*transcript* S48⟩+≡ ◁S76f S77b▷

```
-> 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:

S77b. ⟨*transcript* S48⟩+≡ ◁S77a S77c▷

```
-> 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.

S77c. ⟨*transcript* S48⟩+≡ ◁S77b S77d▷

```
-> [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:

S77d. ⟨*transcript* S48⟩+≡ ◁S77c S77e▷

```
?- [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:

S77e. ⟨*transcript* S48⟩+≡ ◁S77d S77f▷

```
-> [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:

S77f. ⟨*transcript* S48⟩+≡ ◁S77e S78b▷

```
?- 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:

S78a. ⟨*bad version of* `power` S78a⟩≡

```
-> 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 on page S111).

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

S78b. ⟨*transcript* S48⟩+≡ ◁S77f S78c▷

```
?- [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 19 on page S111.

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:

S78c. ⟨*transcript* S48⟩+≡ ◁S78b S78d▷

```
?- [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.

S78d. ⟨*transcript* S48⟩+≡ ◁S78c S78e▷

```
-> 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 | 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 | YS`]` is a permutation of XS if XS can be split into two parts, WS and `[`Y | 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.

S78e. ⟨*transcript* S48⟩+≡ ◁S78d S79a▷

```
-> [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.

S79a. ⟨*transcript* S48⟩+≡ ◁S78e S79b▷

```
?- [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 compared with functional programming, it is easy to code "functions" that want 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:

S79b. ⟨*transcript* S48⟩+≡ ◁S79a S80a▷

```
-> [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 20 on page S111.

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.

S80a. ⟨*transcript* S48⟩+≡ ◁S79b S80b▷

```
?- [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.

S80b. ⟨*transcript* S48⟩+≡ ◁S80a S80c▷

```
-> [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:

S80c. ⟨*transcript* S48⟩+≡ ◁S80b S81▷

```
?- [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]`:

S81. ⟨*transcript* S48⟩+≡ ◁S80c S88▷

```
-> [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

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

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 on page S110).

D.5 Implementation

The implementation of μProlog differs most obviously from our 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, we have 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.

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

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.

S82a. ⟨*µProlog's database of clauses* S82a⟩≡ (S87b)

```
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.1. Substitutions are discovered by solving equality constraints, which are defined here:

S82b. ⟨*substitution and unification* S82b⟩≡ (S87a) S83c ▷

```
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 of terms, goals, clauses S82c⟩
⟨substitutions for µProlog S571a⟩
```

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.

S82c. ⟨*free variables of terms, goals, clauses* S82c⟩≡ (S82b)

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

```
fun termFreevars t =
  let fun f (VAR x,      xs) = insert (x, xs)
        | f (LITERAL _, xs) = 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.5 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.

S83a. ⟨*renaming μProlog variables* S83a⟩≡ (S87a) S83b ▷

`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.

S83b. ⟨*renaming μProlog variables* S83a⟩+≡ (S87a) ◁ S83a

`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.

S83c. ⟨*substitution and unification* S82b⟩+≡ (S87a) ◁ S82b

`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.

S83d. ⟨*constraint solving* **[[prototype]]** S83d⟩≡

`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 on page S68), 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.

[9] Clauses preceding the ⇒ arrow are irrelevant to any future computation, and `search` discards them.

Here is the code:

S84a. ⟨*search* **⟦prototype⟧** S84a⟩≡

```
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 :: S85d⟩ [])
      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
```

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; a "definition" is either a clause or a query.

S84b. ⟨*definitions of* `basis` *and* `processDef` *for μProlog* S84b⟩≡ (S87b)

```
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 S85a⟩; 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.

S85a. ⟨*query goals* gs *against* database S85a⟩≡ (S84b)

```
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, we apply the substitution to the free varables of the query. If we're prompting, we wait for a line of input. If the line begins with a semicolon, we continue; otherwise we quit. If we're not prompting, we're in batch mode, and we produce at most one solution.

S85b. ⟨*interaction* S85b⟩≡ (S87b)

```
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 = separate ("", "\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
```

To make μProlog more compatible with other implementations of Prolog, I patch the `useFile` function defined in Chapter 5. If `useFile` fails with an I/O error, I try adding "`.P`" to the name; this is the convention used by XSB Prolog. If adding `.P` fails, I try adding "`.pl`"; this is the convention used by GNU Prolog and SWI Prolog.

S85c. ⟨*definition of* useFile*, to read from a file* S85c⟩≡

```
val try = useFile
fun useFile filename =
  try filename          handle IO.Io _ =>
  try (filename ^ ".P") handle IO.Io _ =>
  try (filename ^ ".pl")
```

D.5.5 Primitives

This section describes μProlog's handful of primitive predicates, starting with `true`.

S85d. ⟨*μProlog's primitive predicates* :: S85d⟩≡ (S84a) S85e ▷

```
("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.

S85e. ⟨*μProlog's primitive predicates* :: S85d⟩+≡ (S84a) ◁ S85d S86a ▷

```
("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.

S86a. ⟨*µProlog's primitive predicates* :: S85d⟩+≡ (S84a) ◁S85e S86d▷

```
("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.

S86b. ⟨*functions* eval, is, *and* compare, *used in primitive predicates* S86b⟩≡ (S87b) S86c▷

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.

S86c. ⟨*functions* eval, is, *and* compare, *used in primitive predicates* S86b⟩+≡ (S87b) ◁S86b S86e▷

```
fun is [x, e] succ fail = (succ (solve (x ~ LITERAL (eval e))) fail
                           handle Unsatisfiable => fail())
  | is _      _    fail = fail ()
```

S86d. ⟨*µProlog's primitive predicates* :: S85d⟩+≡ (S84a) ◁S86a S86f▷

```
("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.

S86e. ⟨*functions* eval, is, *and* compare, *used in primitive predicates* S86b⟩+≡ (S87b) ◁S86c

```
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.

S86f. ⟨*µProlog's primitive predicates* :: S85d⟩+≡ (S84a) ◁S86d S86g▷

```
("<",  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.

S86g. ⟨*µProlog's primitive predicates* :: S85d⟩+≡ (S84a) ◁S86f

```
("!",   fn _ => raise RuntimeError "The cut (!) must be added to the interpreter") ::
("not", fn _ => raise RuntimeError "Predicate `not' must be added to the interpreter") ::
```

D.5.6 *Putting the pieces together*

The μProlog interpreter is composed of these parts:

S87a. ⟨*upr.sml* S87a⟩≡
 ⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S237a⟩
 ⟨*abstract syntax for* μ*Prolog* S58f⟩
 ⟨*support for tracing* μ*Prolog computation* S583d⟩
 ⟨*substitution and unification* S82b⟩
 ⟨*renaming* μ*Prolog variables* S83a⟩
 ⟨*lexical analysis and parsing for* μ*Prolog, providing* `cqstream` S574c⟩
 ⟨*evaluation, testing, and the read-eval-print loop for* μ*Prolog* S87b⟩
 ⟨*function* `runAs` *for* μ*Prolog* S583b⟩
 ⟨*code that looks at* μ*Prolog's command-line arguments and calls* `runAs` S583c⟩

The evaluation parts are organized as follows:

S87b. ⟨*evaluation, testing, and the read-eval-print loop for* μ*Prolog* S87b⟩≡ (S87a)
 ⟨μ*Prolog's database of clauses* S82a⟩
 ⟨*functions* `eval`, `is`, *and* `compare`, *used in primitive predicates* S86b⟩
 ⟨*tracing functions* S119⟩
 ⟨*search* (left as an exercise)⟩
 ⟨*interaction* S85b⟩
 ⟨*shared definition of* `withHandlers` (left as an exercise)⟩
 ⟨*definitions of* `basis` *and* `processDef` *for* μ*Prolog* S84b⟩
 ⟨*shared unit-testing utilities* S246d⟩
 ⟨*definition of* `testIsGood` *for* μ*Prolog* S572d⟩
 ⟨*shared definition of* `processTests` S247b⟩
 ⟨*shared read-eval-print loop and* `processPredefined` (left as an exercise)⟩

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 on page S88). 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 S90. 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).

[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.

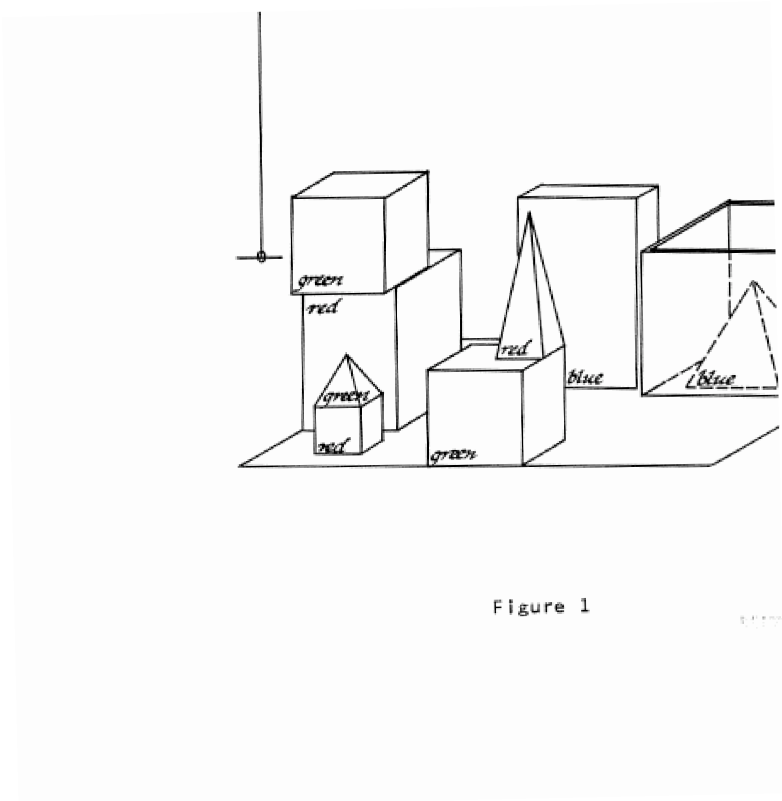


Figure D.4: The original blocks world as depicted by Winograd (1972)

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 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:

S88. ⟨*transcript* S48⟩+≡ ◁ S81 S89a ▷

```
?- [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.

S89a. ⟨*transcript* S48⟩+≡ ◁S88 S89b▷

```
-> 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:

S89b. ⟨*transcript* S48⟩+≡ ◁S89a S89c▷

```
-> 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`)`.

S89c. ⟨*transcript* S48⟩+≡ ◁S89b S89d▷

```
-> 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.

S89d. ⟨*transcript* S48⟩+≡ ◁S89c S89e▷

```
-> 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.

S89e. ⟨*transcript* S48⟩+≡ ◁S89d S90a▷

```
-> 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

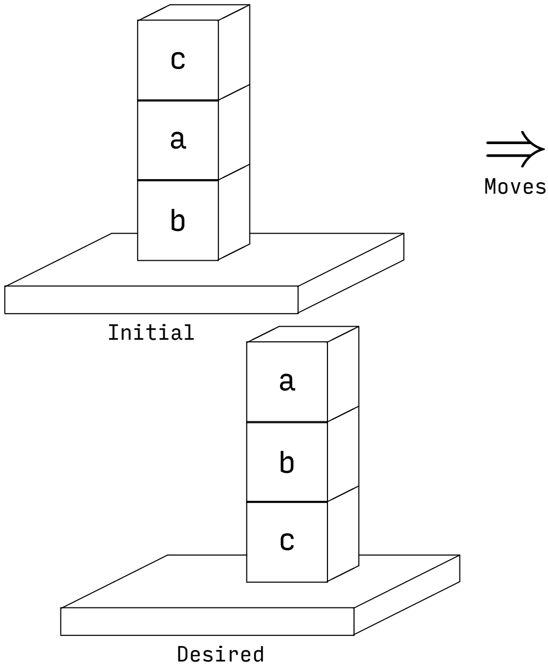


Figure D.5: Example problem in the simplified blocks world

making progress. To rule out these useless moves, here is a predicate that is provable only if a move changes the state.

S90a. ⟨*transcript* S48⟩+≡ ◁S89e S90b▷

```
-> 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.

S90b. ⟨*transcript* S48⟩+≡ ◁S90a S91c▷

```
-> 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:

S90c. ⟨*nonterminating version of* `transforms` S90c⟩≡ S90d▷

```
-> 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:

S90d. ⟨*nonterminating version of* `transforms` S90c⟩+≡ ◁S90c

```
-> 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`:[11]

[11]You can't actually change an existing clause. All you can do is add new clauses to the database. (In full Prolog, you can remove a clause using the fancy predicate `retract`, but let's not go there—it's

S91a. ⟨*nonterminating version of* `transforms`*, with debugging code* S91a⟩≡
```
-> transforms(Initial, [Move|Moves], Final) :-
     good_move(Move, Initial),
     triple(Initial, Move, Intermediate),
     print(moved(Move, Intermediate)),
     transforms(Intermediate, Moves, Final).
```
Now we can see what is going on:

S91b. ⟨*output from nonterminating version of* `transforms`*, with debugging code* S91b⟩≡
```
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. This problem is a classic problem in any connected graph, and it has a classic solution: don't visit the same states repeatedly. The algorithm is depth-first search, and it needs an auxiliary variable to hold the set of states already visited. To hold such a variable in a Prolog program, we create a *4-argument* version of 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`.

S91c. ⟨*transcript* S48⟩+≡ ◁S90b S91d▷
```
-> 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, []).
```
Predicate `not_member` does just what the name says.

S91d. ⟨*transcript* S48⟩+≡ ◁S91c S91e▷
```
-> 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.

S91e. ⟨*transcript* S48⟩+≡ ◁S91d S91f▷
```
-> 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:

S91f. ⟨*transcript* S48⟩+≡ ◁S91e S92a▷
```
-> 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
```

way too far outside the logical interpretation.) What you really do is blow up your interactive session and start over with new definitions.

The plan works, but it's not great. Moving block b twice in a row is not smart. Eliminating double moves helps (Exercise 21 on page S111), 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.

S92a. ⟨*transcript* S48⟩+≡ ◁ S91f S92b ▷

```
?- [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.

S92b. ⟨*transcript* S48⟩+≡ ◁ S92a S92c ▷

```
-> 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:

S92c. ⟨*transcript* S48⟩+≡ ◁ S92b S93a ▷

```
-> [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`.

S93a. ⟨*transcript* S48⟩+≡ ◁S92c S93b▷
```
?- [rule].
-> implemented_by(eq, int, inteq).
```

And integers can be compared for order.

S93b. ⟨*transcript* S48⟩+≡ ◁S93a S93c▷
```
-> 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:

S93c. ⟨*transcript* S48⟩+≡ ◁S93b S93d▷
```
-> 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))
```

S93d. ⟨*transcript* S48⟩+≡ ◁S93c S94a▷
```
-> implemented_by(lt, bool, lambda([arg(p,bool),arg(q,bool)],if(p,false,q))).
```

[12] Warning: at the end of each list, the grammar shows a specious comma.

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 : τ1] [p2 : τ2])
    (if (=1 (fst p1) (fst p2))
        (=2 (snd p1) (snd p2))
        #f))
```

Here it is in Prolog:

S94a. ⟨*transcript* S48⟩+≡ ◁S93d S94b▷

```
-> 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):

S94b. ⟨*transcript* S48⟩+≡ ◁S94a S94c▷

```
-> [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 use some simplification—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.

S94c. ⟨*transcript* S48⟩+≡ ◁S94b S95a▷

```
?- [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:

S95a. ⟨*transcript* S48⟩+≡ ◁S94c S95b▷
```
-> [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)
```
Here's how that rule is coded in μProlog:

S95b. ⟨*transcript* S48⟩+≡ ◁S95a S95c▷
```
?- [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.

S95c. ⟨*transcript* S48⟩+≡ ◁S95b S95d▷
```
-> admits_equality(T) :- implemented_by(eq, T, F).
```
Here, as in ML, types emit equality as long as no function types are involved.

S95d. ⟨*transcript* S48⟩+≡ ◁S95c S98a▷
```
-> [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 S64), 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`:

S98a. ⟨*transcript* S48⟩+≡ ◁ S95d S98b ▷

```
?- [rule].
-> not_equal(X,Y) :- equal(X,Y), !, fail.
-> not_equal(X,Y).
```

where the definition of `equal` is the single clause:

S98b. ⟨*transcript* S48⟩+≡ ◁ S98a

```
-> 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, which 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 S54. 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 on page S120).

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 S121 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.

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

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.

Propositional logic A language of uninterpreted propositions and logical connectives. There are several popular sets of connectives, all equivalent. One minimal set is implication $\implies$ 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.

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.

Object What a variable may stand for in logic; a thing from a (mathematical) domain.

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.

FUNCTOR Prolog's name for an uninterpreted function symbol, expecting one or more arguments.

TERM Prolog's representation of a mathematical object: an atom, a number, or a functor applied to one or more terms.

PROPOSITION The fundamental unit of propositional logic (that is, logic without quantifiers). In Prolog, a PREDICATE applied to zero or more arguments.

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`.

PROPERTY Convenient shorthand for a one-place PREDICATE.

RELATION Convenient shorthand for a PREDICATE of two or more places. Also, the species of mathematical object that a predicate stands for.

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.

GROUND TERM A term that contains no LOGICAL VARIABLES.

SUBSTITUTION A finite mapping from LOGICAL VARIABLES to TERMS. Extends to structure-preserving mappings on terms and CLAUSES.

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.

SUBGOAL A subsidiary GOAL spawned by Prolog's proof search. Also, one conjunct in a goal that is a conjunction.

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.

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 $t_1 \sim t_2$.

FACT A PROPOSITION asserted as fact and entered into the Prolog database. If it contains logical variables, they are implicitly universally quantified.

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.

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.

DIFFERENCE LIST A representation of a list that includes an unbound logical variable, as in `diff([1,2,3|XS], XS)`. Difference lists support many interesting programming techniques; for a good exposition, see Sterling and Shapiro (1986, Chapter 15).

THE CUT An extra-logical feature of Prolog used to limit backtracking and to implement negation. Written as an exclamation mark (`!`). When the cut appears as a premise in a clause, attempts to prove it always succeed, but backtracking into the cut causes the goal from the clause's head to fail—even if there are other clauses that match the goal.

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).

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.

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.

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.

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 on page S109), 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 on page S108 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 on page S112 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 on page S116, 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 on page S120 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.

Guide to all the exercises

Exercises 1 to 3 are warmups. Exercise 1 asks you to prove that Socrates is mortal. Exercise 2 asks you to define two different predicates, both called `mother`, but with different arities. Exercise 3 asks you to define predicates that show who celebrates Mother's Day.

Exercises 4 to 8 build on the map-coloring example in Section D.2. Exercise 4 asks you to color the Atlantic Ocean blue. Exercise 5 asks you to define a new predicate that makes it easier to define maps, and to define and color a new map of Europe. Exercise 6 asks you to color my map of Europe using *four* colors. Exercise 7 asks you to color a map that is represented as an *adjacency list*, not as an inference rule. Exercise 8 asks you to instrument code and work out the rest of the computation that colors the map of the British Isles.

Exercises 9 to 11 are exercises in logic. Exercise 9 asks you to implement Peano's theory of the natural numbers. Exercise 10 asks you to determine when a Boolean formula is satisfied. Exercise 11 asks you to convert a Prolog clause to a Horn clause.

Exercises 12 to 17 are list exercises. Exercise 12 asks you to remove elements from a list. Exercise 13 asks you to split a list into equal parts. Exercise 14 asks you to duplicate the μScheme function `flatten` from Chapter 2, but in a way that can be sometimes run backward—and to use it backward to compute a triangular list. Exercises 15 and 16 ask you to implement insertion sort and merge sort. And Exercise 17 ask you to define some predicates on *difference* lists.

Exercises 18 to 20 explore predicates that can't be run backward or might not always terminate. Exercise 18 asks about `power`; Exercise 19 asks about `fac`; and Exercise 20 asks about `quicksorted`.

Exercise 21 asks you to implement and measure some variations on the move solver for the blocks world.

Exercises 22 to 24 explore some implications of the procedural interpretation of Prolog. Exercise 22 asks you to define `backprint`, a predicate that prints not when you try to prove it, but when you backtrack into it. Exercise 23 asks you to distinguish the procedural interpretation from the logical interpretation by defining two predicates that behave differently only because of a cut. Exercise 24 asks you to use the cut to simplify the definition of `not_equal` from Section D.8.3.

Exercises 25 and 26 ask you to write rules of operational semantics and type systems in Prolog. Exercise 25 asks for an evaluator and Exercise 26 asks for a type checker.

Exercises 27 to 32 are about peg-solitaire puzzles. Exercise 27 asks you to write code that figures out if there is a way to leave at most N pegs on a 10-hole peg-solitaire board. Exercises 28 and 29 ask you to compute the minimum number of pegs that can be left on peg-solitaire boards of 10 holes and 15 holes, respectively. Exercise 30 asks you to compute a sequence of moves that solves peg solitaire, where you can specify in which hole you want the single peg left. Exercise 31 asks you to compute a winning sequence of moves from any starting configuration. Finally, Exercise 32 asks you to solve some of the same problems, but on a peg-solitaire board of arbitrary size—the size of the board becomes another input.

Exercises 33 to 35 present "logic problems," where you are given a bunch of facts about some objects and you have to find the unique relation that is consistent with the facts. "It was Colonel Mustard in the library with the candlestick"; that sort of thing.

Exercises 36 and 37 explore the semantics of Prolog. Exercise 36 asks you to prove facts about substitutions, and Exercise 37 asks you to complete a big-step operational semantics for the procedural interpretation of Prolog (not including the cut).

Exercises 38 to 48 work with the interpreter.

Exercise 38 asks you to implement the constraint solver. Exercise 39 asks you to investigate the consequences of omitting the occurs check in the constraint solver.

Exercise 40 asks you to implement a primitive predicate, and Exercise 41 asks you to prevent anyone from defining a predicate that shares a name with a primitive predicate.

Exercise 42 asks you to improve the usability of the interpreter by adding a tracing facility, and Exercise 43 asks you to improve the performance of the interpreter by changing the representation of the database.

Exercises 44 to 47 ask you to improve μProlog so it is closer to full Prolog. Exercises 44 and 45 asks you implement the cut and the primitive `not` predicate, respectively. Exercise 46 asks you to change the types of primitive predicates so they can look at and modify the database, and Exercise 47 asks you to use this ability to implement `assert` and `retract`.

Finally, Exercise 48 is a companion to Exercise 37: it asks you to reimplement the `query` function in direct style, without streams instead of continuations. It is based on the operational semantics you write in Exercise 37.

D.10.1 Digging into the language

1. Using two clauses and a query, express Aristotle's famous syllogism in Prolog.

2. 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. As I write, her 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. Building on the previous exercise, let us suppose 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. 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. 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. The map of Western Europe, or at least that part shown in Figure D.1 (b), 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.1 (b). Ignore the Atlantic Ocean, the Iberian and Scandinavian peninsulas, and all the other interesting parts of Europe that aren't shown.

7. 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:

S108a. ⟨*exercise transcripts* S108a⟩≡ S108b ▷

```
-> ireland([Atl, Ir, NI], [adj(Atl, [Ir, NI]), adj(Ir, [NI])]).
```

(a) Using the adjacency-list representation, define the predicate `coloring/1`, which is holds if its argument is a properly colored adjacency list. Consider using the predicate `alldifferent/2` from Exercise 5 on page S107.

S108b. ⟨*exercise transcripts* S108a⟩+≡ ◁ S108a S108c ▷

```
-> [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. 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 S71. 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. One of the 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:

S108c. ⟨*exercise transcripts* S108a⟩+≡ ◁ S108b S109 ▷

```
?- print_int(succ(succ(zero))).
2
yes
```

Except for part (f), don't use the primitive `is` predicate.

10. 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.

S109. ⟨*exercise transcripts* S108a⟩+≡ ◁S108c S111▷

```
?- satisfied(and(A, and(B, not(C)))).
A = true
B = true
C = false
yes
```

11. 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. 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. To split lists 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. This exercise explores 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 182.

    (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 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. Implement insertion sort by defining predicate `isorted/2`, where `isorted(`NS`, `MS`)` holds whenever MS is the result of sorting the list of numbers NS.

16. Implement merge sort by defining predicate `msorted/2`, where `msorted(`NS`, `MS`)` holds whenever MS is the result of sorting the list of numbers NS.

17. Program the following operations on difference lists. Don't simply transform them to ordinary lists.

    (a) `diffsnocced`

    (b) `diffreversed`

    (c) `diffquicksorted`

18. 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` S78a⟩ doesn't work.

19. Consider the definition of the predicate `fac` in chunk S78b. 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.

20. Explain why `quicksorted` can't be run backward.

21. These problems concern the blocks-world code:

    (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. 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.

S111. ⟨*exercise transcripts* S108a⟩+≡ ◁S109 S112b▷

```
?- 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. 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. Rewrite the predicate `not_equal` from Section D.8.3 on page S98 so that it still uses the cut, but it does not require the auxiliary predicate `equal`.

25. 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:

$$\frac{}{\langle \text{VAL}(v), \rho\rangle \Downarrow v} \qquad \text{(Constant)}$$

$$\frac{\langle e_1, \rho\rangle \Downarrow v_1 \qquad v_1 = \text{BOOLV}(\texttt{\#t}) \qquad \langle e_2, \rho\rangle \Downarrow v_2}{\langle \text{IF}(e_1, e_2, e_3), \rho\rangle \Downarrow v_2} \qquad \text{(IfTrue)}$$

Let's represent judgment $\langle e, \rho\rangle \Downarrow v$ as the Prolog predicate `eval(`e`, `ρ`, `v`)`. Then we can write these rules:

S112a. ⟨*sample rules for nano-ML evaluation* S112a⟩≡
```
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.

S112b. ⟨*exercise transcripts* S108a⟩+≡ ◁S111
```
?- 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. 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:

S112c. ⟨*sample run of a type checker in Prolog* S112c⟩≡
```
| ?- 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. 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. 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. 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. 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. 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. 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 S99, 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. Definition D.1 on page S60 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. 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 on page S121.

D.10.4 Digging into the interpreter

38. Implement the constraint solver. That is, write function `solve` in chunk S83d. 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 459 of Chapter 7. If you need guidance, Chapter 7 explains constraint solving in detail.

39. 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. 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. 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. Create a tracing version of the interpreter that logs every entry to and exit from a Byrd box. Use the following functions:

S119. ⟨*tracing functions* S119⟩≡ (S87b)

```
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
  )
```

43. 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. 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.

- Change the implementation of function `query` in ⟨*search* **[[prototype]]** S84a⟩ 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. 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. In μProlog, the implementation of a primitive predicate has ML type

$$\forall\alpha.\texttt{term list} \to (\texttt{subst} \to (\texttt{unit} \to \alpha) \to \alpha) \to (\texttt{unit} \to \alpha) \to \alpha.$$

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 $\texttt{unit} \to \alpha$ to $\texttt{database} \to \alpha \times \texttt{database}$.

(b) Change the type of every success continuation from $\texttt{subst} \to (\texttt{unit} \to \alpha) \to \alpha$ to $\texttt{database} \to \texttt{subst} \to (\texttt{database} \to \alpha \times \texttt{database}) \to \alpha \times \texttt{database}$.

(c) Change the type of `query` to

$$\forall\alpha.\texttt{db} \to \texttt{goal list} \to (\texttt{db} \to \texttt{subst} \to (\texttt{db} \to \alpha{\times}\texttt{db}) \to \alpha{\times}\texttt{db}) \to (\texttt{db} \to \alpha{\times}\texttt{db}) \to \alpha{\times}\texttt{db},$$

where `db` is short for `database`.

(d) Change the type of every primitive predicate to

$$\forall\alpha.\texttt{term list} \to (\texttt{db} \to \texttt{subst} \to (\texttt{db} \to \alpha{\times}\texttt{db}) \to \alpha{\times}\texttt{db}) \to (\texttt{db} \to \alpha{\times}\texttt{db}) \to \alpha{\times}\texttt{db},$$

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. Using the interpreter from Exercise 46,

(a) Define primitive predicates `assert` and `retract` as described on page S100.

(b) Test your work by using `assert` to convert a map-coloring *adjacency list* (Exercise 7 on page S107) 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 on page S114).

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 arguments `sick(Patient)`, `psychiatrist(Doctor)`, and `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. Using your operational semantics from Exercise 37 on page S117, 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 I.4.2 on page S249).
- 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

S121. ⟨*streams* S121⟩≡ (S237a) S122a ▷

```
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

S122a. ⟨*streams* S121⟩+≡ (S237a) ◁S121 S122b▷

```
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, but you can get a stream of substitutions using `streamOfCPS (p args)`, where `p` represents the primitive predicate, `args` represents its arguments, and `streamOfCPS` is defined as follows:

S122b. ⟨*streams* S121⟩+≡ (S237a) ◁S122a S122c▷

```
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:

S122c. ⟨*streams* S121⟩+≡ (S237a) ◁S122b

```
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)
```

Chapter contents

VI. Long programming examples

Chapter contents

E Extended programming examples

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, we follow McCarthy's lead, presenting a μScheme interpreter in μScheme. (We interpret just the core of μScheme, without the extended definitions, so there is no implementation of `use`, `check-expect`, `check-assert`, or `check-error`.)

We represent expressions exactly as if they were quoted literals. For example, we represent the expression `(+ x 4)` by the S-expression `'(+ x 4)`.

Our evaluator has much the same structure as the C version, but we use higher-order functions in ways that are not possible in C.

E.1.1 The environment and value store

We represent locations as numbers. The store is an association list from numbers to values, so $\mathrm{dom}\,\sigma = NUM$. To support allocation, the store also maps the special key `next` to a fresh location n. The representation satisfies the invariant that $\forall i \geq n : i \notin \mathrm{dom}\,\sigma$.

S129a. ⟨*eval.scm* S129a⟩≡ S129b ▷

```
(val emptystore '((next 0)))
```

We make the store a global variable `sigma`.

S129b. ⟨*eval.scm* S129a⟩+≡ ◁ S129a S129c ▷

```
⟨definition of find-c generated automatically⟩
(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, we use the special key `'next`. We give `allocate` the same interface as in C.

S129c. ⟨*eval.scm* S129a⟩+≡ ◁ S129b S130a ▷

```
(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.

S130a. ⟨*eval.scm* S129a⟩+≡ ◁ S129c S130b ▷

```
(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, we 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, we can represent most values as themselves. That is, we use symbols to represent symbols, numbers to represent numbers, etc. The exception is functions. Rather than represent each function as itself, we represent every function as a unary function, which takes a list of arguments, possibly changes the store, and returns a single result. We call such a function a "function in list form."

To transform a primitive μScheme function into list form, we define `apply-prim`. We exploit our knowledge that all primitives are either unary or binary.

S130b. ⟨*eval.scm* S129a⟩+≡ ◁ S130a S130c ▷

```
(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)))))))
```

We make 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, we just get the underlying error message from the μScheme interpreter.

E.1.3 The initial environment and store

We can now build the initial environment. We start with an empty `env` and use `let*` to bind each primitive in sequence.

S130c. ⟨*eval.scm* S129a⟩+≡ ◁ S130b S131a ▷

```
(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

We're ready to explore the structure of the evaluator. Because the environment changes only when we make a function call, we define `eval` in curried form. It accepts an environment and returns a function from expressions to values. We call this inner function `ev`.

S131a. ⟨*eval.scm* S129a⟩+≡ ◁S130c S133b▷

```
⟨auxiliary functions for evaluation S131c⟩
(define eval (env)
   (letrec
       ([ev (lambda (e) ⟨result of evaluating expression e in environment env S131b⟩)]
         ⟨letrec bindings of functions used to evaluate abstract syntax S132f⟩)
    ev))
```

Symbols are variables, the locations of which must be looked up in the environment. Other atoms evaluate to themselves.[1] Lists are function applications, unless they are abstract syntax.

S131b. ⟨*result of evaluating expression* `e` *in environment* `env` S131b⟩≡ (S131a)

```
(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 S132a⟩
          ⟨evaluate first to a function, and apply it to arguments from rest S131d⟩))))
```

To find a variable, we use `find-c`, so we can fail if the variable is not found.

S131c. ⟨*auxiliary functions for evaluation* S131c⟩≡ (S131a) S132b▷

```
(define find-variable (x env)
  (find-c x env (lambda (x) x) (lambda () (error (list2 'unbound-variable: x)))))
```

Function application is straightforward. We don't bother to check to see if we are applying a non-function; the underlying μScheme interpreter does that for us. It takes much less space to write the code than to say what it does!

S131d. ⟨*evaluate* `first` *to a function, and apply it to arguments from* `rest` S131d⟩≡ (S131b)

```
((ev first) (map ev rest))
```

[1]The empty list shouldn't evaluate to itself; it should be an error, but we ignore that fine point.

Abstract syntax is a bit more involved. We use brute force to check all the reserved words.

S132a. ⟨*evaluate* `first` *with* `rest` *as abstract syntax* S132a⟩≡ (S131b)

```
(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.

S132b. ⟨*auxiliary functions for evaluation* S131c⟩+≡ (S131a) ◁S131c S132c▷

```
(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)))
```

S132c. ⟨*auxiliary functions for evaluation* S131c⟩+≡ (S131a) ◁S132b S132d▷

```
(define unary (name f rest)
  (if (holds-exactly? rest 1)
    (f (car rest))
    (error (list3 name 'expression-needs-one-argument,-got rest))))
```

S132d. ⟨*auxiliary functions for evaluation* S131c⟩+≡ (S131a) ◁S132c S132e▷

```
(define binary (name f rest)
  (if (holds-exactly? rest 2)
    (f (car rest) (cadr rest))
    (error (list3 name 'expression-needs-two-arguments,-got rest))))
```

S132e. ⟨*auxiliary functions for evaluation* S131c⟩+≡ (S131a) ◁S132d

```
(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`.

S132f. ⟨`letrec` *bindings of functions used to evaluate abstract syntax* S132f⟩≡ (S131a) S132g▷

```
(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.

S132g. ⟨`letrec` *bindings of functions used to evaluate abstract syntax* S132f⟩+≡ (S131a) ◁S132f S132h▷

```
(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. We use `foldl`.

S132h. ⟨`letrec` *bindings of functions used to evaluate abstract syntax* S132f⟩+≡ (S131a) ◁S132g S133a▷

```
(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 we use the real `lambda` to make a closure.

S133a. ⟨`letrec` *bindings of functions used to evaluate abstract syntax* S132f⟩+≡ (S131a) ◁S132h

```
(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.

S133b. ⟨*eval.scm* S129a⟩+≡ ◁S131a S134a▷

```
⟨functions used to evaluate definitions S133c⟩
(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)))
```

The hardest definition to implement is `val`, which must see if the name `x` is already bound in the environment. We examine the environment using function `find-c` from Section 2.10 on page 138. If `x` is bound, we leave `env` alone; otherwise we extend `env` by binding `x` to the empty list. Once `x` is safely bound, we evaluate a `set` expression.

S133c. ⟨*functions used to evaluate definitions* S133c⟩≡ (S133b) S133d▷

```
(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: we rewrite it into a `val` declaration.

S133d. ⟨*functions used to evaluate definitions* S133c⟩+≡ (S133b) ◁S133c S133e▷

```
(define meta-define (env)
  (lambda (name formals body)
    ((meta-val env) name (list3 'lambda formals body))))
```

Since we don't have a `read` primitive, we can't implement `use`. The only other "definition" is evaluation of a top-level expression.

S133e. ⟨*functions used to evaluate definitions* S133c⟩+≡ (S133b) ◁S133d

```
(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.

S134a. ⟨*eval.scm* S129a⟩+≡ ◁S133b S134b▷

```
(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 zero, we make it possible to use `run` interactively without having to look at the final environment and store, which can be quite large.)

S134b. ⟨*eval.scm* S129a⟩+≡ ◁S134a

```
(define run (es)
  (begin (read-eval-print (primenv) es) 0))
```

E.1.7 Tests

These tests exercise functions `apply-prim`, `initialenv`, `meta-lambda`, `eval`, `evaldef`, `meta-if`, `meta-set`, `meta-val`, `meta-define`, `meta-exp`, `read-eval-print`, and `rep`.

S134c. ⟨*evaltest.scm* S134c⟩≡ S134d▷

```
'(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`.

S134d. ⟨*evaltest.scm* S134c⟩+≡ ◁S134c

```
'(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)))
```

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 (1998) 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 S133e 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

   (let ([x_1 e_1] ... [x_n e_n]) e) $\equiv$ ((lambda (x_1 ... x_n) e) e_1 ... e_n)

   You may find `map` more helpful than `foldr`.

   (b) Similarly, add `let*` to the metacircular evaluator using the two laws

   (let* () e) $\equiv$ e
   (let* ([x_1 e_1] ... [x_n e_n]) e) $\equiv$ (let ([x_1 e_1]) (let* (... [x_n e_n[redacted] e))

   As usual, use the standard higher-order functions to help.

   (c) Add `letrec` to the metacircular evaluator by rewriting

   (letrec ([x_1 e_1] ... [x_n e_n]) e)

   to

   (let ([x_1 '()] ... [x_n '()])
     (begin (set x_1 e_1) ... (set x_n e_n) e))

   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:

   (and) $\equiv$ #t
   (and p) $\equiv$ p
   (and p_1 p_2 ... p_n) $\equiv$ (if p_1 (and p_2 ... p_n) #f)

   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:

```
(or)                 ≡ #f
(or e)               ≡ e
(or e1 ··· en)       ≡ (let ([x e1]) (if x x (or e2 ··· en))),
```

where x does not appear in any e_i

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 164b.)

   (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 `al2` = `(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.

   For the measurements in this exercise, use the tests in chunks S134c and S134d.

E.2 Large μML example: 2D-trees

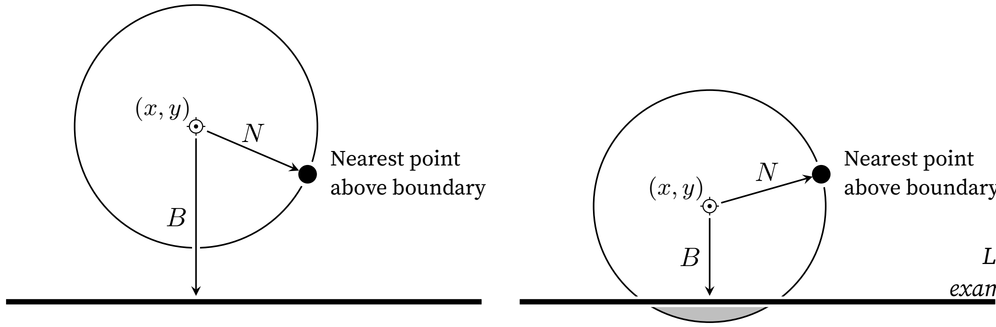
If you want to study full programs that use algebraic data types, this book is full of them: from Chapter 5 onward, every expression and every definition in every interpreter is represented using algebraic data types. But algebraic data types are good for more than just interpreters—they are good representations 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,



$N \leq B$: Needn't search below boundary $N > B$: Must search below boundary

Figure E.1: Search in a 2D-tree (the two important cases)

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 on page S140 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.

- 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 we 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. So we have 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 `2Dpoint`s, like the black dot in Figure E.1. Each point carries an x and y coordinate, plus a value of any type it likes.

S138a. ⟨*gis.uml* S138a⟩≡ S138b ▷

```
(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 this algebraic data type.

S138b. ⟨*gis.uml* S138a⟩+≡ ◁ S138a S138c ▷

```
(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.

S138c. ⟨*gis.uml* S138a⟩+≡ ◁ S138b S138d ▷

```
(check-type 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.

S138d. ⟨*gis.uml* S138a⟩+≡ ◁ S138c S139a ▷

```
(check-type 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`. Each pair of other interesting cases (the rotations) is handled by a different call to `near-or-far`.

S139a. ⟨*gis.uml* S138a⟩+≡ ◁ S138d S140 ▷

```
(check-type nearest-point
            (forall ['a] (int int (2Dtree 'a) -> (2Dpoint 'a))))

(define nearest-point (x y tree)
  (letrec (⟨definition of near-or-far within letrec S139b⟩)
    (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 $N^2 \leq B^2$, we're done; otherwise we search the far subtree and take the closer of the two points.

S139b. ⟨*definition of* `near-or-far` *within* `letrec` S139b⟩≡ (S139a)

```
[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 we 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.

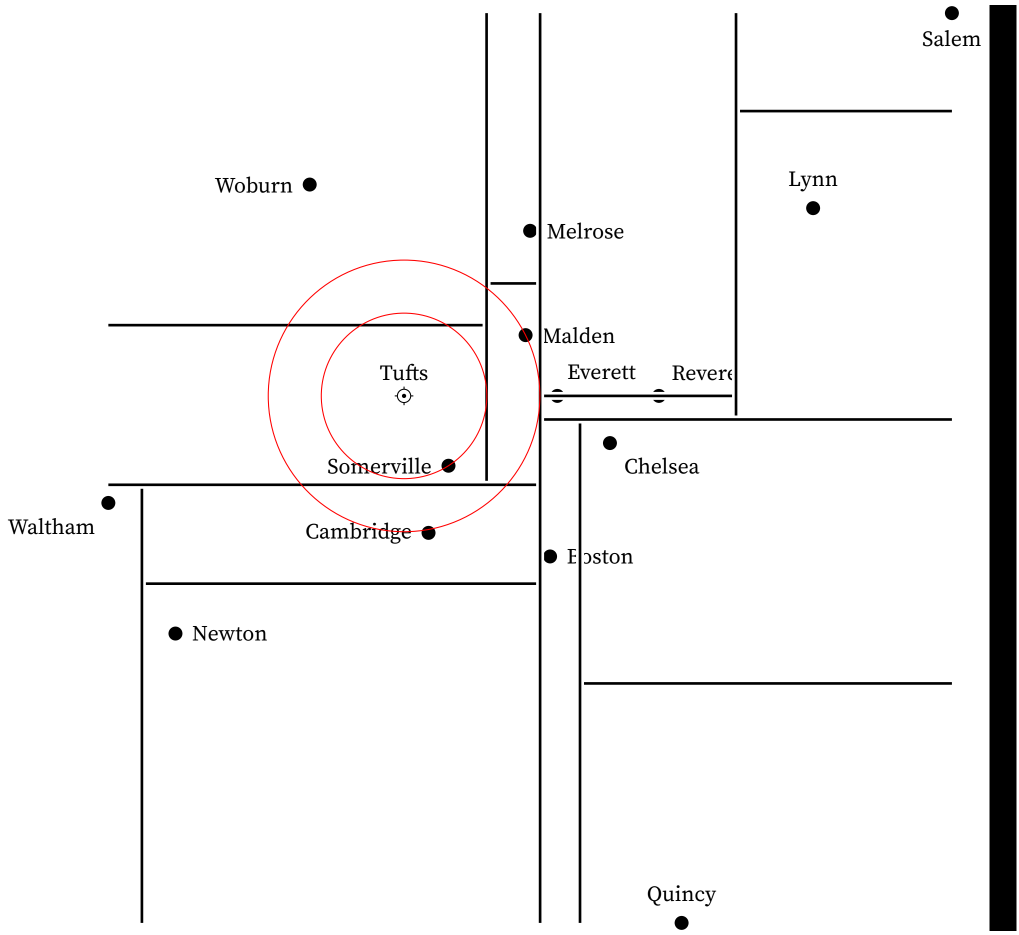


Figure E.2: Balanced 2D-tree of city halls near Boston, searched at Tufts (see page S145)

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.)

S140. ⟨*gis.uml* S138a⟩+≡ ◁S139a S141a▷

```
(check-type mergesort
            (forall ['a] (('a 'a -> order) -> ((list 'a) -> (list 'a)))))
(check-type sort-on ;    sorts on a projection
            (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)))))
```

After sorting a list of points, I split it into halves. Here is the specification of function `halves`: Here is the specification of function `halves`:

$$\texttt{(halves } xs\texttt{)} = \texttt{(pair } ys\ zs\texttt{)},$$
$$\text{where } xs = \texttt{(append } ys\ zs\texttt{)} \text{ and } |\texttt{(length } ys\texttt{)} - \texttt{(length } zs\texttt{)}| \leq 1.$$

S141a. ⟨*gis.uml* S138a⟩+≡ ◁S140 S141b▷

```
(check-type 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, with 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.

S141b. ⟨*gis.uml* S138a⟩+≡ ◁S141a S141c▷

```
(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). Here are auxiliary functions `first` and `last`, which are defined only on nonempty lists.

S141c. ⟨*gis.uml* S138a⟩+≡ ◁S141b S141d▷

```
(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))`.

S141d. ⟨*gis.uml* S138a⟩+≡ ◁S141c S142a▷

```
(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.

S142a. ⟨*gis.uml* S138a⟩+≡ ◁S141d S142b▷

```
(val vert-funs  (make-coord-funs 2Dpoint-x VERT))
(val horiz-funs (make-coord-funs 2Dpoint-y HORIZ))
(check-type vert-funs  (forall ['a] (coord-funs 'a)))
(check-type horiz-funs (forall ['a] (coord-funs 'a)))
```


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.

S142b. ⟨*gis.uml* S138a⟩+≡ ◁S142a S142d▷

```
(check-type 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 S142c⟩])]))])
      (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.

S142c. ⟨*build tree using* cfuns *with* points S142c⟩≡ (S142b)

```
(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)))
```

Here are some rudimentary tests:

S142d. ⟨*gis.uml* S138a⟩+≡ ◁S142b S143a▷

```
(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, we 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."

S143a. ⟨*gis.uml* S138a⟩+≡ ◁S142d S143b▷

```
(record deg ([microdegrees : int]))
```

To compute the difference between two angles, I subtract their microdegrees.

S143b. ⟨*gis.uml* S138a⟩+≡ ◁S143a S143c▷

```
(check-type 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.

S143c. ⟨*gis.uml* S138a⟩+≡ ◁S143b S143d▷

```
(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.

S143d. ⟨*gis.uml* S138a⟩+≡ ◁S143c S144a▷

```
(check-type 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.

S144a. ⟨*gis.uml* S138a⟩+≡ ◁S143d S144b▷

```
(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.

S144b. ⟨*gis.uml* S138a⟩+≡ ◁S144a S144c▷

```
(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:

S144c. ⟨*gis.uml* S138a⟩+≡ ◁S144b S144d▷

```
(define /-round (dividend divisor)
  (/ (+ dividend (/ divisor 2)) divisor))
```

And finally, the conversion functions:

S144d. ⟨*gis.uml* S138a⟩+≡ ◁S144c S144e▷

```
(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, we can convert a point of interest into a proper `2Dpoint` whose x and y coordinates represent distance from Boston in units of `distance-unit-in-meters`.

S144e. ⟨*gis.uml* S138a⟩+≡ ◁S144d S144f▷

```
(check-type 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.

S144f. ⟨*gis.uml* S138a⟩+≡ ◁S144e S144g▷

```
(check-type 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.

S144g. ⟨*gis.uml* S138a⟩+≡ ◁S144f

```
(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))
```

Here is the search shown in Figure E.2 on page S140, except that it uses 83 city halls, not just the fourteen shown in the figure.

S145. ⟨*2D-trees transcript* S145⟩≡

```
-> (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:

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 subtree for Woburn/Somerville and the subtree for 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.

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

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.

S146a. ⟨*exercise transcripts* S146a⟩≡ S146c ▷

```
-> (check-type nearest-point-satisfying
      (forall ['a] (int int ('a -> bool) (2Dtree 'a) -> (option (2Dpoint 'a)))))
```

S146b. ⟨*answers* S146b⟩≡ S146d ▷

```
(use gis.uml)
(val hello 'HELLO)
(define nearest-point-satisfying (x y p? tree)
  (letrec (⟨definition of near-or-far-satisfying within letrec generated automatically⟩)
    (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.

S146c. ⟨*exercise transcripts* S146a⟩+≡ ◁ S146a S147a ▷

```
-> (check-type nearest-k-points
      (forall ['a] (int int int (2Dtree 'a) -> (list (2Dpoint 'a)))))
```

S146d. ⟨*answers* S146b⟩+≡ ◁ S146b

```
(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:

   S147a. ⟨*exercise transcripts* S146a⟩+≡ ◁S146c S147b▷

   ```
   -> (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, and to distinguish the vertical split from the horizontal split, that value constructor takes 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 the `project` function.

   (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

   S147b. ⟨*exercise transcripts* S146a⟩+≡ ◁S147a

   ```
   -> (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.

(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 More examples of Molecule

POSSIBLY TO BECOME EXERCISES!

E.3.1 Bit sets

S148a. ⟨*bitset.mcl* S148a⟩≡

```
(module [Bitset : (exports [abstype t]
                           [empty : t]
                           [insert : (int t -> t)]
                           [inter : (t t -> t)]
                           [union : (t t -> t)]
                           [print : (t -> unit)]
                           [println : (t -> unit)])]
   (type t int)
   (val empty 0)
   (define t insert ([i : t] [s : t]) (Int.lor s (Int.<< 1 i)))
   (val inter Int.land)
   (val union Int.lor)
   (define bool nonzero? ([n : int]) (!= n 0))
   (define unit print ([s : t])
     (Char.print Char.left-curly)
     (let ([i 0])
       (while (< i 32)
          (when (nonzero? (inter s (Int.<< 1 i)))
             (Char.print Char.space)
             (Int.print i))
          (set i (+ i 1))))
     (Char.print Char.space)
     (Char.print Char.right-curly))
   (define unit println ([s : t])
     (print s)
     (Char.print Char.newline))
)
```

E.3.2 Other

E.3.3 Sets of integers, using a stronger invariant

We represent a set of integers as a list with no repeated elements, just as in Section 2.3.7 on page 106. But to improve the cost model, we add a representation invariant: every list is sorted.

S148b. ⟨*int-set.clu* S148b⟩≡

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.

Table E.3: Operations on histograms

```
(cluster int-set [exports [insert   : (int int-set -> int-set)]
                          [member?  : (int int-set -> bool)]
                          [union    : (int-set int-set -> int-set)]
                          [elements : (int-set ->* int)]
                          ⟨other exported operations of cluster int-set (left as an exercise)⟩]
   (type rep int-list) ; invariant: members are strictly increasing
   ⟨operations of cluster int-set S149a⟩
)
```

The new invariant demands changes in some operations and enables changes in others. For example, the `insert` operation must insert into a *sorted* list, without duplicates, so it is almost but not exactly the same as the `insert` function defined in chunk 103a. The `member?` operation does not have to change, but it can be changed so that it doesn't necessarily inspect all the elements: if `n` is bigger than the first element of the representation, then `n` is not in the set.

Where the new invariant really pays off is in the implementation of `union`. To see the payoff, let's start with a naïve implementation of `union`: we compute the union of two sets `nset` and `mset` by inserting each element of `nset` into `mset`.

S149a. ⟨*operations of cluster* `int-set` S149a⟩≡ (S148b) S149b ▷

```
(define naive-union ([nset : int-set] [mset : int-set] -> int-set)
  (for [(n : int)] (elements nset)
     (set mset (insert n mset)))
  (return mset))
```

This naïve implementation of `union` treats both `nset` and `mset` only as abstractions; you can tell because it does not use `unseal`. Such an implementation is correct no matter how sets are represented. But in the worst case, it takes quadratic time. But both `nset` and `mset` are represented by *sorted* lists, and if we inspect both representations, we can implement set union using list merge, which takes linear time:

S149b. ⟨*operations of cluster* `int-set` S149a⟩+≡ (S148b) ◁ S149a S150a ▷

```
(define merge-lists ([ns : int-list] [ms : int-list] -> int-list)
  (if (int-list$null? ns)
    (return ms)
    (if (int-list$null? ms)
      (return ns)
```

```
(begin
  (val [n : int] (int-list$car ns))
  (val [m : int] (int-list$car ms))
  (if (= n m)
    (return (int-list$cons n (merge-lists (int-list$cdr ns) (int-list$cdr ms))))
    (if (< n m)
      (return (int-list$cons n (merge-lists (int-list$cdr ns) ms)))
      (return (int-list$cons m (merge-lists ns (int-list$cdr ms))))))))))
```

An optimized implementation of `union` inspects the representations of both `nset` and `mset`, using `unseal` on both arguments.

S150a. ⟨*operations of cluster* `int-set` S149a⟩+≡ (S148b) ◁S149b

```
(define optimized-union ([nset : int-set] [mset : int-set] -> int-set)
  (return (seal (merge-lists (unseal nset) (unseal mset)))))
```

Completing the implementation of `int-set` and measuring the effect of the optimized union operation is the subject of Exercise 46 on page 613.

E.3.4 Another interface: the histogram

NEED TO ORGANIZE ALL THE EXAMPLES:

- WHAT ARE WE DOING WITH INTERFACES?
- WHAT ARE WE DOING WITH GENERIC MODULES?

Here's another example: the *histogram*. Given these interfaces, we can write and typecheck client code that uses association lists and histograms. The same interface can be used with many different clients.

A histogram, like an array, is a species of finite map from integers to values. In a histogram, the value is always a natural number, intended to represent a thing to be counted. The abstraction is mutable, and it offers the operations shown in Table E.3.

S150b. ⟨*histogram.mcl* S150b⟩≡

```
(module-type HISTOGRAM
   [exports [abstype t]
            [new      : ( -> t)]
            [inc      : (int t -> unit)]
            [inc-by   : (int int t -> unit)]
            [count-of : (int t -> int)]
            [println  : (t -> unit)]])
```

The histogram offers a few benefits over a simple array:

- There's no need to worry about low and high bounds—when a histogram is mutated, bounds are extended as needed.
- A count is incremented in a single operation, instead of a load-modify-store sequence.
- The `println` function offers a simple but pleasant visualization of the contents of a histogram.

I use histograms below to verify the cost model of a hash table.

E.4 Extended μSmalltalk example: Discrete-event simulation

Having been introduced to the μSmalltalk language and its initial basis, we're ready to tackle a more ambitious example. The example in this section is big enough that you can see some interplay among classes and methods. This sort of interplay is characteristic of object-oriented programs. In this example, we look at a problem faced by our 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 his 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 we 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: we don't know if the simulation models what would really happen. In this section, we don't worry about realism; our goal is to learn Smalltalk.

In the robot lab, the interesting events happen at discrete points 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 we explore below.

E.4.1 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 our 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 our 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.
  - It keeps track of whatever information about the simulation is important, so when the simulation is over, it can report conclusions. In our 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]

S152. ⟨*simulation classes* S152⟩≡ S154a ▷

```
(val ActiveSimulation nil)
```

[2]In Smalltalk-80, `ActiveSimulation` would be a class variable (page 711), not a global variable.

Instance protocol for `Simulation`:

`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.)

Global variable used by `Simulation`:

`ActiveSimulation`	Holds the value of the currently active `Simulation` object.

Figure E.4: Partial instance protocol for class `Simulation`

Using this design, you can expect most of a simulation to be programmed with 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.

Instance protocol for PriorityQueue:

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: Protocol for class PriorityQueue

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.4.2 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, we 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 10.17 on page 659. (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.)
- 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.

S154a. ⟨*simulation classes* S152⟩+≡ ◁S152 S156c▷

```
(class Simulation
    [subclass-of Object]
    [ivars now eventQueue]
    (method time-now () now)
    ⟨more methods of class Simulation S154b⟩
)
```

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 S167.

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).

S154b. ⟨*more methods of class* Simulation S154b⟩≡ (S154a) S155a▷

```
  (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`.

S155a. ⟨*more methods of class* `Simulation` S154b⟩+≡ (S154a) ◁S154b S155b▷

```
  (method finishUp ()
      (set ActiveSimulation nil)
      self)
```

The `proceed` method simulates the next event in the queue.

S155b. ⟨*more methods of class* `Simulation` S154b⟩+≡ (S154a) ◁S155a S155c▷

```
  (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 S170.)

We 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 we use to run robot-lab simulations.

S155c. ⟨*more methods of class* `Simulation` S154b⟩+≡ (S154a) ◁S155b S155d▷

```
  (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, we 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.

S155d. ⟨*more methods of class* `Simulation` S154b⟩+≡ (S154a) ◁S155c S155e▷

```
  (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." We schedule an event by using the `at:put:` method of class `PriorityQueue` to add the event to the event queue.

S155e. ⟨*more methods of class* `Simulation` S154b⟩+≡ (S154a) ◁S155d S156a▷

```
  (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."

S156a. ⟨*more methods of class* `Simulation` S154b⟩+≡ (S154a) ◁S155e S156b▷

```
(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, we define a new class of simulation object which is called `RecurringEvents`. An object of class `RecurringEvents` is initialized with an `eventFactory` and a `timeStream`.

S156b. ⟨*more methods of class* `Simulation` S154b⟩+≡ (S154a) ◁S156a

```
(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.

S156c. ⟨*simulation classes* S152⟩+≡ ◁S154a S157▷

```
(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 other methods (class method `new:atNextTimeFrom:` and instance method `init:with:`) implement the common pattern, first shown in Section 10.1, in which we create an object by sending a message to a class method, which then uses an instance method to initialize the new object.

E.4.3 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 we 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 we 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.

S157. ⟨*simulation classes* S152⟩+≡ ◁S156c S158a▷

```
(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:

S158a. ⟨*simulation classes* S152⟩+≡ ◁S157 S159e▷

```
(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 S158b⟩
)
```


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, we pass three parameters: a time limit t, a student factory `s`, and a stream of interarrival times `as`.

S158b. ⟨*methods of class* `RobotLabSimulation` S158b⟩≡ (S158a) S158c▷

```
(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.

S158c. ⟨*methods of class* `RobotLabSimulation` S158b⟩+≡ (S158a) ◁S158b S159a▷

```
(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`:

S159a. ⟨*methods of class* RobotLabSimulation S158b⟩+≡ (S158a) ◁ S158c S159b ▷

```
(class-method new () (self error: 'robot-lab-simulation-needs-arguments))
```

Our finishUp method reports on the results of the simulation. We print just the information we 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.

S159b. ⟨*methods of class* RobotLabSimulation S158b⟩+≡ (S158a) ◁ S159a S159c ▷

```
(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))
```

At entry and exit, the simulation updates its internal statistics:

S159c. ⟨*methods of class* RobotLabSimulation S158b⟩+≡ (S158a) ◁ S159b S159d ▷

```
(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.

S159d. ⟨*methods of class* RobotLabSimulation S158b⟩+≡ (S158a) ◁ S159c S160a ▷

```
(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 we 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.

S159e. ⟨*simulation classes* S152⟩+≡ ◁ S158a S160b ▷

```
(class Queue
    [subclass-of List]
)
```

Instance protocol for `Student`:

`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.

Private methods 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.

Class protocol 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.

Figure E.6: Protocol for `Student`

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.

S160a. ⟨*methods of class* `RobotLabSimulation` S158b⟩+≡ (S158a) ◁S159d

```
(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.

The `Student` class represents a student by six instance variables.

S160b. ⟨*simulation classes* S152⟩+≡ ◁S159e S163d▷

```
(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
```

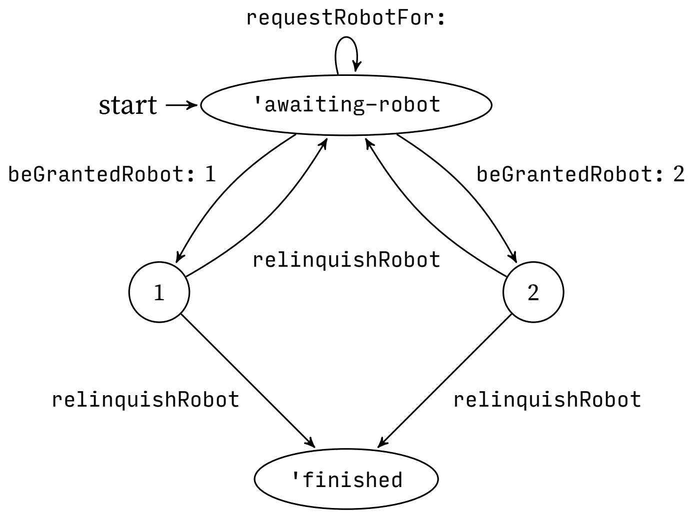


Figure E.7: State-transition diagram for a `Student`

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`

```
      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 S162a⟩
)
```

Here are some notes on the use of these instance variables:

- 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.
- Variable `timeNeeded` holds total amount of time the student needs with the robot in order to finish his or her lab work. Variable `timeStillNeeded` holds the amount of time left after whatever time the student has already spent with the robot. Our 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 we're trying to gather. The data is provided to the simulation by the `timeWaiting` method.

S162a. ⟨*other methods of class* Student S162a⟩≡ (S160b) S162b ▷

```
(method timeWaiting ()
    (exitTime - (entryTime + timeNeeded)))
```

To create a Student object, we 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. But there's a little something extra going on with timeNeeded:

S162b. ⟨*other methods of class* Student S162a⟩+≡ (S160b) ◁ S162a S162c ▷

```
(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 ofinstance 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 it is told—it will receive a takeAction message from the RobotLabSimulation. Its action depends on its status.

S162c. ⟨*other methods of class* Student S162a⟩+≡ (S160b) ◁ S162b S163a ▷

```
(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 he needs more time, he puts himself in the 'awaiting-robot state, and he immediately requests the robot again. (He'll either wait in the queue, or in the special case where nobody else is waiting, he'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, he notes the current time as the exitTime from the simulation, and then he exits 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`.

S163a. ⟨*other methods of class* `Student` S162a⟩+≡ (S160b) ◁S162c S163b▷

```
(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.

S163b. ⟨*other methods of class* `Student` S162a⟩+≡ (S160b) ◁S163a S163c▷

```
(method needsRobot? () (timeStillNeeded > 0))
```

The last remaining action in the `Student` class shows what happens when a student is granted use of a robot. He or she keeps 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.

S163c. ⟨*other methods of class* `Student` S162a⟩+≡ (S160b) ◁S163b

```
(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.4.4 Running robot-lab simulations

To create a robot-lab simulation, we need a time limit, a student class, and a stream of interarrival times. We can then run the simulation for any given number of minutes. In a serious simulation, we would put a lot of effort into the classes that represent students' needs and arrival times. We 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.

Our first simplifying assumption is that every student needs two hours of robot time, which we measure in minutes:

S163d. ⟨*simulation classes* S152⟩+≡ ◁S160b S164a▷

```
(class Student120 [subclass-of Student] ; a student needing 120 minutes of robot time
    (method timeNeeded () 120)
)
```

Our second simplifying assumption is that we have 20 students, and they all pour into the lab the moment it opens (i.e., when the simulation starts). We need to embody this assumption as an infinite stream of interarrival times. In other words, we 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:

S164a. ⟨*simulation classes* S152⟩+≡ ◁S163d S164c▷

```
(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}))
)
```

We use these classes, plus our implementation of PriorityQueue from Exercise 1, to create a simulation sim30. We then run the simulation for 20 simulated hours:

S164b. ⟨*simulation transcript* S164b⟩≡ S165b▷

```
-> (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 we want to minimize average waiting time, we 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 we create a new Student, we'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.

S164c. ⟨*simulation classes* S152⟩+≡ ◁S164a S165a▷

```
(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 we 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, we won't cap the number of students at 20; instead, we 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:`.

S165a. ⟨*simulation classes* S152⟩+≡ ◁S164c

```
(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, we 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, we run it four times:

S165b. ⟨*simulation transcript* S164b⟩+≡ ◁S164b

```
-> (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, we shouldn't draw any conclusions.

E.4.5 Summary and analysis

Our simulation omits too many details. For example, a real student who enters the lab and finds a long line may *balk*, i.e., he may leave and try again later. We 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, our simulations make bogus assumptions about needs and about arrival times—and these assumptions probably have a decisive effect on the results. We might build into the simulation a list of needs and arrival times obtained by observing real students, or we might simply invent a probabilistic model that we 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 our 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 we 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, we 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.

[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.

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.

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—though difficult—is the key to becoming a productive object-oriented programmer.

E.4.6 Robot-lab exercises

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.4) 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.

The next group of problems build on the discrete-event simulation of the robot lab, which is described in Section E.4 on page S151.

1. The discrete-event simulation requires a priority queue, whose protocol is given in Figure E.5 on page S154. Use the variable-size arrays from Exercise 23 on page 728 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 we're implementing a priority queue, we can do better than $O(n)$ time for insertion. You can implement a faster algorithm if you store the queue's elements in an array which is indexed from 1 to n and which satisfies the following invariant:

$$\forall k . a[k] \leq a[2k] \wedge a[k] \leq a[2k+1],$$

whenever $2k \le n$ and $2k + 1 \le n$.

(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. There are a number of ways we could improve the robot-lab simulation.

(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: the 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 his or her *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: examples include luggage carts, Boeing 747s, gallons of gasoline, and twenty-dollar bills. There is no reason that every new simulation class should have to implement code to manage fungible resources—it should be done once in the 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 we have a new model of students' needs, we 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.4.2 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.

VII. Interesting infrastructure

Chapter contents

F

Code for writing interpreters in C

Chapter 1 presents only those parts of the Impcore interpreter that are most relevant to the study of programming languages. If that code is the tip of the iceberg, 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, when compared to other techniques I use, it requires elaborate code and complicated data structures. This 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 here is reusable. If you choose to reuse 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 on page S176. A higher-level overview, which shows what information is presented in each chapter or appendix, is shown in Table F.2 on page S177.

F.1 Streams

The 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, 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

Abstract syntax, names, values, functions, and environments

Concept	Types & Functions	Interface	Implementation
Abstract syntax	`Exp`, `Def`	§1.6.1 (pages 43 & 42)	(exposed rep)
Abstract syntax	`XDef`, `UnitTest`	§K.1 (page S288)	(exposed rep)
Names	`Name`	§1.6.1 (page 43)	§K.1.5 (page S293)
Value	`Value`	§1.6.1 (page 44)	(exposed rep)
Function	`Func`, `Userfun`	§1.6.1 (pages 42 & 44)	(exposed rep)
Environment	`Valenv`	§1.6.1 (page 44)	§1.6.3 (page 55)
Environment	`Funenv`	§1.6.1 (page 44)	§K.5 (page S300)

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 54)
Evaluator	`readevalprint`	§K.1 (page S289)	§K.1.3 (page S291)
Interaction	`Echo`	§K.1 (page S289)	(exposed rep)

Streams and lists

Concept	Types & Functions	Interface	Implementation
Extended definitions	`XDefstream`, `filexdefs`, `stringxdefs`, `getxdef` `xdefstream`	§§F.1.3 and K.1 (pages S186 & S288)	§F.1.3 (XDef-stream.impgetxdef.imp)
Parenthesized phrases	`Par`, `Parstream`, `getpar`	§F.1.2 (page S181)	§F.1.2 (page S182)
Lines	`Linestream`, `getline_`	§F.1.1 (page S178)	§F.1.1 (pages S178 & S180)
Lists of `Exp`s, `Value`s, and others	(not shown)	§1.6.1 (page 46)	(generated automatically)

Printing and error signaling

Concept	Types & Functions	Interface	Implementation
Printers	`print`, `fprint`	§1.6.1 (page 46)	§F.3.1 (page S190)
Error-signaling printers	`synerror`, `runerror`, `othererror`	§§1.6.1 and K.1 (page S289 and page 47)	§F.4.1 (page S194)
Error helpers	`checkargc`, `duplicatename`	§§1.6.1 and F.4.2 (page 48 and page S196)	§F.4.2 [redacted]ages S195 & S196)
Printer extension	`installprinter`, `Printer`	§§F.3 and K.1 (pages S189 & S289)	§F.3.2 (page S191)
Source locations	`Sourceloc`	§K.1 (page S289)	(exposed rep)
Error formats	`ErrorFormat`	§K.1 (page S289)	(exposed rep)
Error modes	`ErrorMode`, `set_error_mode`	§F.4 (page S193)	§F.4.1 (page S193)

Table F.1: Key ideas, their interfaces, and their implementations (excludes parsing)

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 `Linestreams` 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

Table F.2: The implementation of Impcore, as organized into chapters, appendices, and files 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.1.1 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. You 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.

S178a. ⟨*shared type definitions* S178a⟩≡ (S290) S181b ▷

```
typedef struct Linestream *Linestream;
```

S178b. ⟨*shared function prototypes* S178b⟩≡ (S290) S178c ▷

```
char *getline_(Linestream r, const char *prompt);
```

To create a `Linestream`, you need a string or a file. And when creating a `Linestream`, you name the source; that name is used in error messages.

S178c. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁ S178b S181d ▷

```
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.

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.

S178d. ⟨*shared structure definitions* S178d⟩≡ (S290)

```
struct Linestream {
    char *buf;              /* holds the last line read */
    int bufsize;               /* size of buf */
```

[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.

```
    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.

S179a. ⟨*linestream.c* S179a⟩≡ S179b ▷

```
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 S179c⟩
    lines->s = s;
    return lines;
}
```

S179b. ⟨*linestream.c* S179a⟩+≡ ◁ S179a S179d ▷

```
Linestream filelines(const char *filename, FILE *fin) {
    Linestream lines = calloc(1, sizeof(*lines));
    assert(lines);
    lines->source.sourcename = filename;
    lines->fin = fin;
    return lines;
}
```

S179c. ⟨*check to see that* s *is empty or ends in a newline* S179c⟩≡ (S179a)

```
{   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.

S179d. ⟨*linestream.c* S179a⟩+≡ ◁ S179b S180a ▷

getline_ S180a

```
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 hidden trick: 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.

S180a. ⟨*linestream.c* S179a⟩+≡ ◁S179d

```
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 S180b⟩
    else if (lines->s)
        ⟨set lines->buf to next line from string lines->s, or return NULL if lines are exhausted S180c⟩
    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, I call 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, I grow the buffer and call `fgets` again, to get more of the line. This process iterates until the last character in the buffer is a newline. I then chop off the newline by overwriting it with `'\0'`.

S180b. ⟨*set* `lines->buf` *to next line from file* `lines->fin`*, or return* `NULL` *if lines are exhausted* S180b⟩≡ (S180a)

```
{
    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';
}
```

When reading from a string, I look in `lines->s`. I find the next newline, copy the characters into `buf`, and update `lines->s`.

S180c. ⟨*set* `lines->buf` *to next line from string* `lines->s`*, or return* `NULL` *if lines are exhausted* S180c⟩≡ (S180a)

```
{
    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.1.2 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.

Here's the definition:

S181a. ⟨*par.t* S181a⟩≡
```
Par* = ATOM (Name)
     | LIST (Parlist)
```

S181b. ⟨*shared type definitions* S178a⟩+≡ (S290) ◁S178a S181c▷
```
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.

S181c. ⟨*shared type definitions* S178a⟩+≡ (S290) ◁S181b S186d▷
```
typedef struct Parstream *Parstream;
```

To create a `Parstream`, you specify not only the lines from which `Par`s will be read, but also the prompts to be used (page S288). To get a `Par` from a stream, call `getpar`. And for error messages, code can ask a `Parstream` for its current source location.

S181d. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S178c S181e▷
```
Parstream parstream(Linestream lines, Prompts prompts);
Par       getpar   (Parstream r);
Sourceloc parsource(Parstream pars);
```

The final part of the interface to a `Parstream` is the global variable `read_tick_as_quo` If `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.

`getpar`	S183a
`growbuf`	S179d
type `Linestream`	S178a
type `Par`	$\mathcal{A}$
`parsource`	S182c
`parstream`	S182b
`print`	46c
type `Prompts`	S288g
type `Sourceloc`	S289d

S181e. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S181d S186e▷
```
extern bool read_tick_as_quote;
```

In Impcore, a tick mark is not read as `(quote ...)`, so `read_tick_as_quote` is false.

S181f. ⟨*impcore.c* S181f⟩≡
```
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.

S182a. ⟨*lex.c* S182a⟩≡ S182b ▷

```
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;
};
```

To create a Parstream, I initialize the fields using the parameters. Initializing input to an empty string puts the stream into a state with no characters left over.

S182b. ⟨*lex.c* S182a⟩+≡ ◁ S182a S182c ▷

```
Parstream parstream(Linestream lines, Prompts prompts) {
    Parstream pars = malloc(sizeof(*pars));
    assert(pars);
    pars->lines = lines;
    pars->input = "";
    pars->prompts.ps1 = prompts == STD_PROMPTS ? "-> " : "";
    pars->prompts.ps2 = prompts == STD_PROMPTS ? "   " : "";
    return pars;
}
```

Function parsource grabs the current source location out of the Linestream.

S182c. ⟨*lex.c* S182a⟩+≡ ◁ S182b S182d ▷

```
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.

I deal with this distinction by writing getpar_in_context, which knows whether it is the first call or another call. Function getpar 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.

S182d. ⟨*lex.c* S182a⟩+≡ ◁ S182c S183a ▷

⟨*prototypes of private functions that help with* getpar S184b⟩

```
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 S183b⟩
        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 S184c⟩
        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 S183c⟩
            } else {
                ⟨read and return an ATOM S184a⟩
            }
        }
    }
}
```

With this code in hand, getpar is a first call.

S183a. ⟨*lex.c* S182a⟩+≡ ◁S182d S184d▷

```
Par getpar(Parstream pars) {
    assert(pars);
    return getpar_in_context(pars, true, '\0');
}
```

To scan past whitespace, I use the standard C library function `isspace`. That function requires an *unsigned* character.

S183b. ⟨*advance* `pars->input` *past whitespace characters* S183b⟩≡ (S182d)

```
while (isspace((unsigned char)*pars->input))
    pars->input++;
```

When `getpar` 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))`).

S183c. ⟨*read a* `Par` *and return that* `Par` *wrapped in* `quote` S183c⟩≡ (S182d)

```
{
    pars->input++;
    Par p = getpar_in_context(pars, false, '\'');
    if (p == NULL)
        synerror(parsource(pars), "premature end of file after quote mark");
    assert(p);
```

type Linestream	S178a
type Par	$\mathcal{A}$
parsource	S181d
type Parstream	S181c
type Prompts	S288g
type Sourceloc	S289d
synerror	48a

```
    return mkList(mkPL(mkAtom(strtoname("quote")), mkPL(p, NULL)));
}
```

Atoms are delegated to function `readatom`, defined below.

S184a. ⟨*read and return an* ATOM S184a⟩≡ (S182d)

```
return mkAtom(readatom(&pars->input));
```

S184b. ⟨*prototypes of private functions that help with* getpar S184b⟩≡ (S182d) S184e ▷

```
static Name readatom(const char **ps);
```


Reading and returning a parenthesized list After a left parenthesis, I read `Par`s until I see a right parenthesis, adding each one to the front of `elems_reversed`. When I get to the closing right parenthesis, I reverse the elements in place and return the resulting list.

S184c. ⟨*read and return a parenthesized* LIST S184c⟩≡ (S182d)

```
{
    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));
}
```

To reverse a list, I use a classic trick of imperative programming: I update the pointers in place. The invariant is exactly the same as the invariant of `revapp` in Section 2.3.2 on page 101. But the code in Section 2.3.2 allocates new memory; the code here only updates pointers, without allocating.

S184d. ⟨*lex.c* S182a⟩+≡ ◁ S183a S185a ▷

```
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;
}
```

S184e. ⟨*prototypes of private functions that help with* getpar S184b⟩+≡ (S182d) ◁ S184b S185d ▷

```
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.

S185a. ⟨*lex.c* S182a⟩+≡ ◁S184d S185b▷
```
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.

S185b. ⟨*lex.c* S182a⟩+≡ ◁S185a S185c▷
```
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.

S185c. ⟨*lex.c* S182a⟩+≡ ◁S185b S185e▷
```
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);
}
```

S185d. ⟨*prototypes of private functions that help with* `getpar` S184b⟩+≡ (S182d) ◁S184e S185f▷
```
static int  isdelim(char c);
static Name strntoname(const char *s, int n);
```

S185e. ⟨*lex.c* S182a⟩+≡ ◁S185c
```
static bool brackets_match(char left, char right) {
    switch (left) {
        case '(': return right == ')';
        case '[': return right == ']';
        case '{': return right == '}';
        default: assert(0);
    }
}
```

S185f. ⟨*prototypes of private functions that help with* `getpar` S184b⟩+≡ (S182d) ◁S185d
```
static bool brackets_match(char left, char right);
```

`mkAtom`	$\mathcal{A}$
`mkList`	$\mathcal{A}$
`type Name`	43b
`type Par`	$\mathcal{A}$
`type Parlist`	S181b
`pars`	S182d
`parsource`	S181d
`type Parstream`	S181c
`strtoname`	43c
`synerror`	48a

F.1.3 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.

S185g. ⟨*xdefstream.c* S185g⟩≡ S186a▷
```
struct XDefstream {
    Parstream pars;                  /* where input comes from */
};
```

[2] In C++, I would instead pass the pointer by reference.

To make an XDefstream, allocate and initialize.

S186a. ⟨*xdefstream.c* S185g⟩+≡ ◁S185g S186b▷

```
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 Parstreams exist. It builds XDefstreams by calling filexdefs or stringxdefs. Those functions build XDefstreams by combining xdefstream and parstream with either filelines or stringlines, respectively.

S186b. ⟨*xdefstream.c* S185g⟩+≡ ◁S186a S186c▷

```
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), NO_PROMPTS));
}
```

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.

S186c. ⟨*xdefstream.c* S185g⟩+≡ ◁S186b

```
XDef getxdef(XDefstream xdr) {
    Par p = getpar(xdr->pars);
    if (p == NULL)
        return NULL;
    else
        return parsexdef(p, parsource(xdr->pars));
}
```

F.2 Buffering characters

A classic abstraction: the resizeable buffer. Function bprint writes to a buffer.

S186d. ⟨*shared type definitions* S178a⟩+≡ (S290) ◁S181c S189b▷

```
typedef struct Printbuf *Printbuf;
```

A buffer is created with printbuf and destroyed with freebuf.

S186e. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S181e S186f▷

```
Printbuf printbuf(void);
void freebuf(Printbuf *);
```

We append to a buffer with bufput or bufputs, and we empty the buffer with bufreset.

S186f. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S186e S186g▷

```
void bufput(Printbuf, char);
void bufputs(Printbuf, const char*);
void bufreset(Printbuf);
```

We can do two things with the contents of a buffer: copy them in to a freshly allocated block of memory, or write them to an open file handle.

S186g. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S186f S188f▷

```
char *bufcopy(Printbuf);
void fwritebuf(Printbuf buf, FILE *output);
```

F.2.1 *Implementation of a print buffer*

This classic data structure needs no introduction.

S187a. ⟨*printbuf.c* S187a⟩≡ S187b ▷

```
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.

S187b. ⟨*printbuf.c* S187a⟩+≡ ◁ S187a S187c ▷

```
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;
}
```

We free a buffer using Hanson's (1996) indirection trick.

S187c. ⟨*printbuf.c* S187a⟩+≡ ◁ S187b S187d ▷

```
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.

S187d. ⟨*printbuf.c* S187a⟩+≡ ◁ S187c S187e ▷

```
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;
}
```

We write a character, at `buf->next`, growing if needed.

S187e. ⟨*printbuf.c* S187a⟩+≡ ◁ S187d S188a ▷

```
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;
}
```

To write a string, we grow until we can call memcpy.

S188a. ⟨*printbuf.c* S187a⟩+≡ ◁S187e S188b▷

```
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;
}
```

To discard all the characters, bufreset.

S188b. ⟨*printbuf.c* S187a⟩+≡ ◁S188a S188c▷

```
void bufreset(Printbuf buf) {
    assert(buf && buf->next);
    buf->next = buf->chars;
}
```

To use the buffer, we want to know how many characters are in it.

S188c. ⟨*printbuf.c* S187a⟩+≡ ◁S188b S188d▷

```
static int nchars(Printbuf buf) {
    assert(buf && buf->chars && buf->next);
    return buf->next - buf->chars;
}
```

Copy a buffer to a fresh block.

S188d. ⟨*printbuf.c* S187a⟩+≡ ◁S188c S188e▷

```
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;
}
```

Write a buffer's characters to an open file handle.

S188e. ⟨*printbuf.c* S187a⟩+≡ ◁S188d

```
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.3 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. My solution is to define new, custom print functions that know how to print values and expressions:

S188f. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S186g S189a▷

```
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, you announce a new format specifier with installprinter, and you provide a function used to print a value so specified.

S189a. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S188f S189d▷

```
void installprinter(unsigned char specifier, Printer *take_and_print);
```

The function provided has type Printer. Its specification is that it takes one value out of the list args, then prints the value to the given buffer.

S189b. ⟨*shared type definitions* S178a⟩+≡ (S290) ◁S186d

```
⟨definition of va_list_box S189c⟩
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:

S189c. ⟨*definition of* va_list_box S189c⟩≡ (S189b)

```
typedef struct va_list_box {
  va_list ap;
} va_list_box;
```

I encourage you to 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:

  S189d. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S189a S191c▷

  ```
  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.

bprint	S190a
grow	S187d
installprinter	S191b
type Printbuf	S186d
vbprint	S191a

[3]Library functions such as vfprintf itself are grandfathered; only *users* cannot write functions that take va_list arguments. Feh.

- The next bricks define `bprint`, `print`, and `fprint` on top of `vbprint`.
- There are a whole bunch of bricks of type `Printfun`: one for each conversion specifier we know how to print (there's a list in Table 1.6 on page 47).

In Section F.4.1 on page S194 below, functions `runerror`, `othererror`, and `synerror` rest on this stack of bricks as well.

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).

F.3.1 *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.

S190a. ⟨*print.c* S190a⟩≡ S190b ▷

```
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.

S190b. ⟨*print.c* S190a⟩+≡ ◁ S190a S190c ▷

```
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.

S190c. ⟨*print.c* S190a⟩+≡ ◁ S190b S191a ▷

```
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.3.2 *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]`.

S191a. ⟨*print.c* S190a⟩+≡ ◁S190c S191b▷

```
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.

S191b. ⟨*print.c* S190a⟩+≡ ◁S191a S191d▷

```
void installprinter(unsigned char c, Printer *take_and_print) {
    printertab[c] = take_and_print;
}
```

F.3.3 *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.

S191c. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S189d S192d▷

```
Printer printpercent, printstring, printdecimal, printchar, printname, printpointer;
```

As in standard `vprintf`, the conversion specifier `%%` just prints a percent sign, without consuming any arguments.

S191d. ⟨*print.c* S190a⟩+≡ ◁S191b S192a▷

```
void printpercent(Printbuf output, va_list_box *box) {
    (void)box;
    bufput(output, '%');
}
```


`bufput`	S186f
`bufputs`	S186f
`bufreset`	S186f
`freebuf`	S186e
type `Printbuf`	S186d
`printbuf`	S186e
`printchar`	S192c
`printdecimal`	S192a
type `Printer`	S189b
`printname`	S192b
`printpointer`	S192a
`printstring`	S192a
type `va_list_box`	S189c
`vbprint`	S189d

The printers for strings and numbers are textbook examples of how to use va_arg.

S192a. ⟨*print.c* S190a⟩+≡ ◁S191d
```
void printstring(Printbuf output, va_list_box *box) {
    const char *s = va_arg(box->ap, char*);
    bufputs(output, s);
}

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);
}

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.

S192b. ⟨*printfuns.c* S192b⟩≡ S192c▷
```
void printname(Printbuf output, va_list_box *box) {
    Name np = va_arg(box->ap, Name);
    bufputs(output, np == NULL ? "<null>" : nametostr(np));
}
```

S192c. ⟨*printfuns.c* S192b⟩+≡ ◁S192b S192e▷
```
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:

S192d. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S191c S193a▷
```
Printer printpar;
```

S192e. ⟨*printfuns.c* S192b⟩+≡ ◁S192c
```
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. Here is a snapshot of what that code might look like:

```
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.4 ERROR FUNCTIONS

The interface in Section 1.6.1 on page 47 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 should *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`.

S193a. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S192d S195b▷

```
typedef enum ErrorMode { NORMAL, TESTING } ErrorMode;
void set_error_mode(ErrorMode mode);
extern jmp_buf testjmp;    /* if error occurs during a test, longjmp here */
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 `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.4.1 *Implementation of error signaling*

The state of the error module includes the error mode and the two `jmp_buf`s.

S193b. ⟨*error.c* S193b⟩≡ S193c▷

```
jmp_buf errorjmp;
jmp_buf testjmp;

static ErrorMode mode = NORMAL;
```

`bprint`	S188f
`bufput`	S186f
`bufputs`	S186f
type `Name`	43b
type `Par`	$\mathcal{A}$
type `Printbuf`	S186d
type `Printer`	S189b
type `va_list_box`	S189c

Function `set_error_mode` sets the error mode.

S193c. ⟨*error.c* S193b⟩+≡ ◁S193b S194a▷

```
void set_error_mode(ErrorMode new_mode) {
  assert(new_mode == NORMAL || new_mode == TESTING);
  mode = new_mode;
}
```

Function `runerror`'s behavior depends on the mode:

- In normal mode, it prints a message, then jumps to `errorjmp`.
- In testing mode, it *buffers* the message, then silently jumps to `testjmp`.

S194a. ⟨*error.c* S193b⟩+≡ ◁S193c S194b▷

```
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.

S194b. ⟨*error.c* S193b⟩+≡ ◁S194a S195a▷

```
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.

S195a. ⟨*error.c* S193b⟩+≡ ◁S194b S195c▷

```
void set_toplevel_error_format(ErrorFormat new_format) {
  assert(new_format == WITH_LOCATIONS || new_format == WITHOUT_LOCATIONS);
  toplevel_error_format = new_format;
}
```


Function `othererror` generalizes `runerror`

S195b. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S193a S196a▷

```
void othererror (const char *fmt, ...);
```

S195c. ⟨*error.c* S193b⟩+≡ ◁S195a S195d▷

```
Printbuf errorbuf;
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);
    }
}
```

`bufcopy`	S186g
`bufreset`	S186f
`errorbuf`	S193a
`type ErrorFormat`	S289e
`errorjmp`	47
`type Exp`	$\mathcal{A}$
`freebuf`	S186e
`longjmp`	$\mathcal{B}$
`mode`	S193b
`type Printbuf`	S186d
`printbuf`	S186e
`runerror`	47
`type Sourceloc`	S289d
`testjmp`	S193a
`type va_list_box`	S189c
`vbprint`	S189d

F.4.2 *Implementations of error helpers*

As promised in Section 1.6.1 on page 48, here are 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.

S195d. ⟨*error.c* S193b⟩+≡ ◁S195c S196b▷

```
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.

S196a. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S195b S197a▷

```
Name duplicatename(Namelist names);
```

S196b. ⟨*error.c* S193b⟩+≡ ◁S195d

```
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 any good C compiler will identify and optimize a direct tail recursion like this one.)

F.5 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:

S196c. ⟨*tests.c* S196c⟩≡

```
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;
```

```
    }
}
```

F.6 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.

S197a. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S196a S198b▷

```
extern int  checkoverflow(int limit);
extern void reset_overflow_check(void);
```

I assume that the stack grows downward.

S197b. ⟨*overflow.c* S197b⟩≡

```
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) {
```

`type Name`	43b
`type Namelist`	43b
`runerror`	47

```
  eval_fuel = default_eval_fuel;
}
```

Here's an example of a detected overflow:

S198a. ⟨*transcript* S198a⟩≡ S198e ▷

```
-> (define blowstack (n) (+ 1 (blowstack (- n 1))))
-> (blowstack 0)
Run-time error: recursion too deep
```


F.7 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 our 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.

S198b. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁ S197a S199a ▷

```
extern void checkarith(char operation, int32_t n, int32_t m, int precision);
```

Only addition, subtraction, multiplication, and division can cause overflow.

S198c. ⟨*arith.c* S198c⟩≡

```
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 S198d⟩
}
```

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.

S198d. ⟨*if* `result` *cannot be represented using* `precision` *signed bits, signal overflow* S198d⟩≡ (S198c)

```
assert(precision > 0 && precision < 64);  // shifts are defined
if ((result << (64-precision)) >> precision != result) {
  runerror("Arithmetic overflow");
}
```

Here's an example of arithmetic overflow:

S198e. ⟨*transcript* S198a⟩+≡ ◁ S198a

```
-> (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
```

F.8 Unicode support

Unicode is a standard that attempts to describe all the world's character sets. In Unicode, a 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.

Here's how we print Unicode characters.

S199a. ⟨*shared function prototypes* S178b⟩+≡ (S290) ◁S198b

```
void fprint_utf8(FILE *output, unsigned code_point);
void print_utf8 (unsigned u);
```

This encoder supports code points of up to 21 bits.

S199b. ⟨*unicode.c* S199b⟩≡ S199c▷

```
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);
    }
}
```

runerror 47

S199c. ⟨*unicode.c* S199b⟩+≡ ◁S199b

```
void print_utf8(unsigned code_point) {
    fprint_utf8(stdout, code_point);
}
```

Chapter contents

G

Parsing parenthesized phrases (including Impcore) in C

A key step in the implementation of any programming language is the translation from the concrete syntax that appears in the input to the abstract syntax of the language in question. 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, we use a different model: sequences of lines are turned into *parenthesized phrases* (Section F.1.2), and these phrases are what is parsed into abstract syntax. The details are the subject of this chapter.

The implementation of a parser, although interesting, is not central to what I hope you get out of this book. Parsing is an art and a science all its own, and it is the subject of its own learned textbooks. Using parenthesized phrases enables us to avoid the usual challenges and complexities. In their place, however, we 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. In 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 on page S217.

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.

S202a. ⟨*shared function prototypes* S202a⟩≡ (S290) S202b ▷

```
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 we are parsing an `Exp` or an `XDef`.

- If the input is an `ATOM`, we are parsing 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.

  S202b. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁ S202a S204c ▷

  ```
  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 components, all of which are expresssions, 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.

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.1: Parsing table for Impcore expressions

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*

Table G.2: Parsing table for Impcore extended definitions

- 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.)

With these properties in mind, here is my plan:

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

`type Exp` $\mathcal{A}$
`exp_of_atom`,
in Impcore S212c
in μScheme (in GC?!) S318a
`type Name` 43b
`type Par` $\mathcal{A}$
`parseexp` S212b
`parsexdef` S213d
`type Sourceloc` S289d
`type XDef` $\mathcal{A}$

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 parser 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 `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.

S204a. ⟨*tableparsing.c* S204a⟩≡ S207a ▷
⟨*private function prototypes for parsing* S209b⟩

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.

S204b. ⟨*structure definitions for Impcore* S204b⟩≡ (S290)

```
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 4.9 on page 292) will detect the error.

The standard reduce functions are `reduce_to_exp` and `reduce_to_xdef`. The first argument codes for what kind of node the components should be reduced to; the second argument points to an *array* that holds the components.

S204c. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁ S202b S207b ▷

```
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:

S205a. ⟨*parse.c* S205a⟩≡ S205d ▷

```
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 S205b⟩
    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* S205b⟩.

S205b. ⟨*cases for Impcore's* `reduce_to_exp` *added in exercises* S205b⟩≡ (S205a) S218a ▷

```
/* 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 we 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, we 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

In the table, *alt* stands for an enumeration literal of the sort to go in an `alt` field.

To enable the codes to appear as cases in `switch` statements, I define them using C macros:

S205c. ⟨*macro definitions used in parsing* S205c⟩≡ (S290)

```
#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.

S205d. ⟨*parse.c* S205a⟩+≡ ◁ S205a S212a ▷

```
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 88, 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 syntax we are trying to parse, the location where it came from, and if there's a keyword or function name involved, what it is.

S206a. ⟨*shared structure definitions* S206a⟩≡ (S290) S210d ▷

```
#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 we are parsing */
        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`.

S206b. ⟨*shared type definitions* S206b⟩≡ (S290) S207c ▷

```
typedef struct ParserState *ParserState;
typedef struct ParsingContext *ParsingContext;
```

When we create a new parser state, all we know is what `Par` we're trying to parse. That gives us the input and part of the context. The output is empty.

S207a. ⟨*tableparsing.c* S204a⟩+≡ ◁S204a S208a▷

```
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;
}
```

S207b. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S204c S207e▷

```
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:

S207c. ⟨*shared type definitions* S206b⟩+≡ (S290) ◁S206b S207d▷

```
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 `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`.

S207d. ⟨*shared type definitions* S206b⟩+≡ (S290) ◁S207c S211b▷

```
typedef ParserResult (*ShiftFun)(ParserState);
```

Here are the four basic shift functions.

S207e. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S207b S208b▷

```
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 */
```


type Name	43b
type Par	$\mathcal{A}$
type Parlist	S181b
sExp	S208c
sExps	S208d
sName	S208f
sNamelist	S209a
type Sourceloc	S289d

The names are abbreviated because *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 we don't need those techniques just yet. For now, let's just implement shift functions.

The shift operation itself is implemented in 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:

S208a. ⟨*tableparsing.c* S204a⟩+≡ ◁ S207a S208c ▷

```
void halfshift(ParserState s) {
    assert(s->input);
    s->input = s->input->tl;
    assert(s->nparsed < MAXCOMPS);
}
```

S208b. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁ S207e S208e ▷

```
void halfshift(ParserState state); /* advance input, check for room in output */
```

Here's a full shift for an expression. It calls parseexp, with which it is mutually recursive.

S208c. ⟨*tableparsing.c* S204a⟩+≡ ◁ S208a S208d ▷

```
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.

S208d. ⟨*tableparsing.c* S204a⟩+≡ ◁ S208c S208f ▷

```
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.

S208e. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁ S208b S208g ▷

```
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.

S208f. ⟨*tableparsing.c* S204a⟩+≡ ◁ S208d S209a ▷

```
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.

S208g. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁ S208e S209d ▷

```
Name parsename(Par p, ParsingContext context);
```

A `Namelist` appears in parenthesis and is used only in the `define` form.

S209a. ⟨*tableparsing.c* S204a⟩+≡ ◁S208f S209c▷
```
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);
    }
}
```


S209b. ⟨*private function prototypes for parsing* S209b⟩≡ (S204a) S211f▷
```
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 what components are expected to be part of a syntactic form. (This technique of using functions as data is developed at length in Chapter 2.) To parse a syntactic form, I call the functions in sequence. As an end-of-sequence marker, I use the function `stop`. It 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`.

S209c. ⟨*tableparsing.c* S204a⟩+≡ ◁S209a S209e▷
```
ParserResult stop(ParserState state) {
    if (state->input == NULL)
        return STOP_PARSING;
    else
        return INPUT_LEFTOVER;
}
```

S209d. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S208g S209f▷
```
ParserResult stop(ParserState state);
```

Finally, I have a special shift function that doesn't do any shifting. Instead, it sets the context for parsing a function definition. Right after calling `sName` with the function name, I call `setcontextname`.

S209e. ⟨*tableparsing.c* S204a⟩+≡ ◁S209c S210a▷
```
ParserResult setcontextname(ParserState s) {
    assert(s->nparsed > 0);
    s->context.name = s->components[s->nparsed-1].name;
    return PARSED;
}
```

S209f. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S209d S210c▷
```
ParserResult setcontextname(ParserState state);
```

Exercise 30 asks you to add local variables to Impcore. Shift function `sLocals` looks for the keyword `locals`. If found, the keyword marks a list of the names of local variables. 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.

S210a. ⟨*tableparsing.c* S204a⟩+≡ ◁S209e S210e▷

```
ParserResult sLocals(ParserState s) {
    Par p = s->input ? s->input->hd : NULL;  // useful abbreviation
    if (⟨Par p represents a list beginning with keyword locals S210b⟩) {
        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;
    }
}
```

Parsing parenthesized phrases in C

S210

The keyword test is just complicated enough that it warrants being put in a named code chunk.

S210b. ⟨`Par p` *represents a list beginning with keyword* `locals` S210b⟩≡ (S210a)

```
p != NULL && p->alt == LIST && p->list != NULL &&
p->list->hd->alt == ATOM && p->list->hd->atom == strtoname("locals")
```

S210c. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S209f S211a▷

```
ParserResult sLocals(ParserState state);  // shift locals if (locals x y z ...)
```

G.4 Representing and parsing tables and rows

As shown in Tables G.1 and G.2 on page S203, 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`.

S210d. ⟨*shared structure definitions* S206a⟩+≡ (S290) ◁S206a

```
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.

S210e. ⟨*tableparsing.c* S204a⟩+≡ ◁S210a S211c▷

```
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);
        }
    }
}
```

S211a. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S210c S211d▷

```
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.

S211b. ⟨*shared type definitions* S206b⟩+≡ (S290) ◁S207d S217a▷

```
typedef struct ParserRow *ParserTable;
```

S211c. ⟨*tableparsing.c* S204a⟩+≡ ◁S210e S211e▷

```
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] S211g⟩
    rowparse(&t[i], s);
    return &t[i];
}
```

S211d. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S211a S214b▷

```
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`.

S211e. ⟨*tableparsing.c* S204a⟩+≡ ◁S211c S212b▷

```
static bool rowmatches(struct ParserRow *row, Name first) {
    return row->keyword == NULL || strtoname(row->keyword) == first;
}
```

S211f. ⟨*private function prototypes for parsing* S209b⟩+≡ (S204a) ◁S209b S216c▷

```
static bool rowmatches(struct ParserRow *row, Name first);
```

Once a row has matched, what we do with it depends on whether it was a `NULL` match or a keyword match. If row `t[i]` has a keyword, then the first `Par` in the input is that keyword, and it needs to be consumed—so we adjust `s->input`. And we set the context.

S211g. ⟨*adjust the state* `s` *so it's ready to start parsing using row* `t[i]` S211g⟩≡ (S211c)

```
if (t[i].keyword) {
    assert(first != NULL);
    s->input = s->input->tl;
    s->context.name = first;
}
```

halfshift	S208b
type Name	43b
type Par	$\mathcal{A}$
parsenamelist	S209b
type ParserResult	S207c
type ParserState	S206b
type ParsingContext	S206b
type ShiftFun	S207d
strtoname	43c
synerror	48a
usage_error	S215d

G.5 Parsing tables and functions

Every language has two parsing tables: one for expressions and one for extended definitions.

S211h. ⟨*declarations of global variables used in lexical analysis and parsing* S211h⟩≡ (S290) S215b▷

```
extern struct ParserRow exptable[];
extern struct ParserRow xdeftable[];
```

Here, as promised from Table G.1 on page S203, 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.

S212a. ⟨*parse.c* S205a⟩+≡ ◁S205d S212c▷

```
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 S213a⟩

struct ParserRow exptable[] = {
  { "set",   SET,    setshifts },
  { "if",    IFX,    ifshifts },
  { "while", WHILEX, whileshifts },
  { "begin", BEGIN,  beginshifts },
  ⟨rows added to Impcore's exptable in exercises S213b⟩
  { NULL,    APPLY,  applyshifts }  /* must come last */
};
```

And here is the corresponding parsing function. The parsing function delegates the heavy lifting to other functions: `exp_of_atom` deals with atoms, and `tableparse` and `reduce_to_exp` deal with lists.

S212b. ⟨*tableparsing.c* S204a⟩+≡ ◁S211e S213c▷

```
Exp parseexp(Par p, Sourceloc source) {
    switch (p->alt) {
    case ATOM:
        ⟨if p->atom is a reserved word, call synerror with source S215a⟩
        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 resued with different versions of `exp_of_atom`, `exptable`, and `reduce_to_exp`.

In Impcore, `exp_of_atom` classifies each atom as either an integer literal or a variable.

S212c. ⟨*parse.c* S205a⟩+≡ ◁S212a S215c▷

```
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.

S213a. ⟨*arrays of shift functions added to Impcore in exercises* S213a⟩≡ (S212a) S218b ▷

```
/* for each new row added to exptable, add an array of shift functions here */
```

S213b. ⟨*rows added to Impcore's* `exptable` *in exercises* S213b⟩≡ (S212a) S218c ▷

```
/* add a row here for each new syntactic form of Exp */
```

Next, here are the parsing table and function for 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`.

S213c. ⟨*tableparsing.c* S204a⟩+≡ ◁ S212b S213d ▷

```
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 S218e⟩
    { NULL,           ADEF(EXP),          expshifts }  /* must come last */
};
```

S213d. ⟨*tableparsing.c* S204a⟩+≡ ◁ S213c S214a ▷

```
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 `EXP` node has just one component, an `Exp`. I can't use `sExp` here, because that consumes just a single item from the input, as an `Exp`. What I need is to treat the *entire* input as an `Exp`. Shift function `use_exp_parser` does the work. This function ignores `s->input`; instead it uses `s->context.par`, which gets passed to `parseexp`.

S214a. ⟨*tableparsing.c* S204a⟩+≡ ◁S213d S214c▷

```
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;
}
```


S214b. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S211d S217b▷

```
ParserResult use_exp_parser(ParserState state);
```

Whenever I expect a name, I actually parse a full expression. Then, if it isn't a name, I complain. This technique allows maximum latitude in case the programmer makes a mistake. The error-handling function `name_error` is described below.

S214c. ⟨*tableparsing.c* S204a⟩+≡ ◁S214a S214d▷

```
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 the two main parsing functions, there are others. A list of expressions is parsed recursively.

S214d. ⟨*tableparsing.c* S204a⟩+≡ ◁S214c S214e▷

```
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, with context information in case of an error.

S214e. ⟨*tableparsing.c* S204a⟩+≡ ◁S214d S215d▷

```
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`.

S215a. ⟨*if* `p->atom` *is a reserved word, call* `synerror` *with* `source` S215a⟩≡ (S212b)

```
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 input with 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.

S215b. ⟨*declarations of global variables used in lexical analysis and parsing* S211h⟩+≡ (S290) ◁S211

```
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[];
```

S215c. ⟨*parse.c* S205a⟩+≡ ◁S212c S217d▷

```
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 S218d⟩
    { -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.

S215d. ⟨*tableparsing.c* S204a⟩+≡ ◁S214e S216a▷

```
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:
```

type `Exp`	$\mathcal{A}$
type `Explist`, in Impcore	S288c
in μScheme (in GC?!)	S303b
`halfshift`	S208b
`mkEL`	$\mathcal{A}$
`mkNL`	$\mathcal{A}$
type `Name`	43b
`name_error`	S216c
type `Namelist`	43b
type `Par`	$\mathcal{A}$
type `Parlist`	S181b
`parseexp`	S202a
`parseexplist`	S208e
`parsenamelist`	S209b
type `ParserResult`	S207c
type `ParserState`	S206b
type `ParsingContext`	S206b
type `Sourceloc`	S289d
`synerror`	48a

```
                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 a name was expected but we saw something else instead, the parser calls `name_error`. The error message says more about what went wrong and what the context is. To make extending `name_error` as easy as possible, I first convert the offending name to an integer code, so that the proper code can be chosen using a `switch` statement.

S216a. ⟨*tableparsing.c* S204a⟩+≡ ◁S215d S216b▷

```
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);
    }
}
```

To discover the proper code, function `code_of_name` does a reverse lookup in `exptable` and `xdeftable`.

S216b. ⟨*tableparsing.c* S204a⟩+≡ ◁S216a

```
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);
}
```

S216c. ⟨*private function prototypes for parsing* S209b⟩+≡ (S204a) ◁S211f

```
void *name_error(Par bad, struct ParsingContext *context);
                        /* expected a name, but got something else */
```

Here are integer codes for all the syntactic forms that are suggested to be implemented as syntactic sugar.

S217a. ⟨*shared type definitions* S206b⟩+≡ (S290) ◁S211b

```
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

S217b. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S214b S217c▷

```
int code_of_name(Name n);
```

In Impcore, there are no expressions that bind names, so expressions need not be checked; only `define` needs to be checked.

S217c. ⟨*shared function prototypes* S202a⟩+≡ (S290) ◁S217b

```
void check_exp_duplicates(Sourceloc source, Exp e);
void check_def_duplicates(Sourceloc source, Def d);
```

The operational semantics requires that in every function definition, the names of the formal parameters be distinct.

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{DEFINE}(f, \langle x_1, \ldots, x_n \rangle, e), \xi, \phi \rangle \rightarrow \langle \xi, \phi\{f \mapsto \text{USER}(\langle x_1, \ldots, x_n \rangle, e)\} \rangle} \quad (\text{DEFINEFUNCTION})$$

I implement this check here, in the parser, so if there's an error, I can give the source-code location.

S217d. ⟨*parse.c* S205a⟩+≡ ◁S215c

```
void check_exp_duplicates(Sourceloc source, Exp e) {
    (void)source; (void)e;
}
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);
}
```

check_def_duplicates S326c
check_exp_duplicates S326c
type Def $\mathcal{A}$
duplicatename S196a
type Exp $\mathcal{A}$
type Name 43b
type Par $\mathcal{A}$
type Sourceloc S289d
synerror 48a

G.7 Extending Impcore with syntactic sugar

Design for extension is all very well, but examples are even better. In this section I add short-circuit `&&` and `||` operators, like those found in C. Unlike the functions `and` and `or`, the syntactic operators `&&` and `||` don't always evaluate all their arguments. For example, in code chunk ⟨`Par p` *represents a list beginning with keyword* `locals` S210b⟩, 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:

$$(\texttt{\&\&}\ e_1\ e_2) \triangleq (\texttt{if}\ e_1\ e_2\ \texttt{0})$$
$$(\texttt{||}\ e_1\ e_2) \triangleq (\texttt{if}\ e_1\ 1\ e_2\ )$$

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."

S218a. ⟨*cases for Impcore's* reduce_to_exp *added in exercises* S205b⟩+≡ (S205a) ◁S205b

```
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);
```

The components of a short-circuit conditional are the two subexpressions e_1 and e_2, so I need an array of shift functions that shifts two expressions and then stops.

S218b. ⟨*arrays of shift functions added to Impcore in exercises* S213a⟩+≡ (S212a) ◁S213a

```
static ShiftFun conditionalshifts[] = { sExp, sExp, stop };
```

The exptable rows use the given shift functions, and the usage_table entries show the expected syntax.

S218c. ⟨*rows added to Impcore's* exptable *in exercises* S213b⟩+≡ (S212a) ◁S213b

```
{ "&&", SUGAR(CAND), conditionalshifts },
{ "||", SUGAR(COR),  conditionalshifts },
```

S218d. ⟨*Impcore* usage_table *entries added in exercises* S218d⟩≡ (S215c)

```
{ SUGAR(CAND), "(&& exp exp)" },
{ SUGAR(COR),  "(|| exp exp)" },
```

The conditional sugar doesn't require any new definition forms.

S218e. ⟨*rows added to* xdeftable *in exercises* S218e⟩≡ (S213c)

```
/* add new forms for extended definitions here */
```

Finally, here is a short demonstration showing how && and || differ from and and or:

S218f. ⟨*transcript* S218f⟩≡

```
-> (|| 1 (println 99))
1
-> (or 1 (println 99))
99
1
-> (&& 0 (println 33))
0
-> (|| 0 (println 33))
33
33
```

components	S205a
mkIfx	$\mathcal{A}$
mkLiteral	$\mathcal{A}$
or	27a
sExp	S207e
type ShiftFun	S207d
stop	S209d

Chapter contents

Supporting discriminated unions in C

H

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 format of the descriptions, which is inspired the Zephyr Abstract Syntax Description Language (Wang et al. 1997), is like this:

S221. ⟨*example input* S221⟩≡
```
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;
    };
};
```

H.1 Lexical analysis

There are a few reserved symbols, a token in all upper case is a constructor, and anything else is a name. Constructors, like ML constructors, identify the alternatives in a sum type.

S222a. ⟨*lexical analysis for μASDL* S222a⟩≡ (S222b S234c) S222c ▷

```
datatype pretoken
  = RESERVED of char
  | CONSTR   of name (* constructor *)
  | NAME     of name
type token = pretoken plus_brackets
```

Conversion to strings is typical.

S222b. ⟨*definitions of type* `token` *and function* `tokenString` *for μASDL* S222b⟩≡

```
⟨lexical analysis for μASDL S222a⟩
fun pretokenString (RESERVED c) = str c
  | pretokenString (NAME n)     = n
  | pretokenString (CONSTR c)   = c
```

`pretokenString : pretoken -> string`

The lexer converts a string to a sequence of tokens. Unlike the other languages in this book, this input language 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.

S222c. ⟨*lexical analysis for μASDL* S222a⟩+≡ (S222b S234c) ◁ S222a

`asdlToken : token lexer`

```
val asdlToken =
  let fun validate NONE = NONE
        | validate (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)
        <|> (validate o streamGet)
  in  whitespace *> bracketLexer token
  end
```

H.2 Abstract syntax and parsing

There are two kinds of definitions: sums and products. The left-hand side of a definition gives a name and lets us know if the thing being defined is a pointer.

S222d. ⟨*abstract syntax for μASDL* S222d⟩≡ (S234d)

`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
```

Our problem domain (generating C) 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. To insert separators, we use a utility function we call `foldr1`. Function `foldr1` is a bit like the standard `foldr`, except that it inserts a binary operator *between* elements of a list. If a list contains a single element, `foldr1` returns that element unchanged. If a list is empty, and only then, `foldr1` uses its second argument.

S223a. ⟨*parsers for μASDL* S223a⟩≡ (S234c) S223b ▷

```
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)
```

Our first use of `foldr1` will be to take a sequence of tokens like `char *` or `char *name` and turn the sequence into a string where adjacent tokens are separated by spaces. This problem is part of our first parsing function, which takes a sequence of tokens and turns it into a field. Because we permit a lone field to be anonymous, we use a heuristic to turn the sequence into a "pre-argument," which is like an `arg` except that it may not be named.

S223b. ⟨*parsers for μASDL* S223a⟩+≡ (S234c) ◁ S223a S223c ▷

```
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. The function is Curried so that we can partially apply it, then pass the result to `map`.

S223c. ⟨*parsers for μASDL* S223a⟩+≡ (S234c) ◁ S223b S223d ▷

```
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.

S223d. ⟨*parsers for μASDL* S223a⟩+≡ (S234c) ◁ S223c S223f ▷

```
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
```

S223e. ⟨*utility functions for string manipulation and printing* S223e⟩≡

```
val lower = String.map Char.toLower
val upper = String.map Char.toUpper
```

Finally, our parser:

S223f. ⟨*parsers for μASDL* S223a⟩+≡ (S234c) ◁ S223d

```
name : name    parser
alt  : alt     parser
arg  : pre_arg parser
def  : def     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 res c = eqx c reservedChar

fun commas p = curry op :: <$> p <*> many (res #"," *> p)
fun bars   p = curry op :: <$> p <*> many (res #"|" *> p)

fun leftRound tokens =
  let fun check (_, ROUND) = OK ROUND
        | check (loc, shape) =
            errorAt ("don't use " ^ leftString shape ^ "; use (") loc
  in  (check <$>! left) tokens
  end

fun product (args : pre_arg list) =
  errorList (map (nameRequired "defined type") args) >>=+ PRODUCT

val arg   = preArg <$>! (many (name <|> "*" <$ res #"*"))
val type' = pair <$> name <*> ((fn t => {ptr = isSome t}) <$> optional (res #"*"))
val args  = leftRound <&> bracket ("(arg, ...)", commas (arg <?> "arg"))
val alt   = toAlt <$> constructor <*>! optional args
val def   = pair <$> type' <*> (res #"=" *> (product <$>! args <|> SUM <$> bars alt))
```

H.3 Interface to a general-purpose prettyprinter

We want to generate C code with reasonable indentation and line breaks. Laying out text with suitable indentation and line breaks is called *prettyprinting*. The problem has a long history (Oppen 1980; Hughes 1995; Wadler 1999). 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`. The most basic documents are formed from strings. Subdocuments may be concatenated (`^^`) to form larger documents, and subdocuments may also be indented. (Indentation is relative to surrounding documents.) Finally, the creator of the document controls *exactly* where a line break may be introduced: the `BREAK` indicates that a break is permissible, but if the break is not taken, the prettyprinter inserts the selected string instead.

S224a. ⟨*algebraic laws for the prettyprinting combinators* S224a⟩≡ S224b ▷

```
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')
```

```
type doc
doc     : string -> doc
^^      : doc * doc -> doc
empty   : doc
brk     : doc
indent : int * doc -> doc
empty   : doc
```

There are also laws relating to layout:

S224b. ⟨*algebraic laws for the prettyprinting combinators* S224a⟩+≡ ◁ S224a

```
layout (d ^^ d')         = layout d ^ layout d'
layout empty             = ""
```

```
layout : int -> doc -> string
```

```
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 our convenience we add a line-breaking concatenate (`^/`) and some support for adding breaks and semicolons:

S225a. ⟨*prettyprinting combinators* S225a⟩≡ (S234d)

```
⟨definition of doc and functions S232a⟩
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
```

H.4 C TYPES

The main C types we are interested in 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 we emit 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.

S225b. ⟨*C types* S225b⟩≡ (S234d) S226a ▷

```
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 we can emit `typedef`s:

S226a. ⟨*C types* S225b⟩+≡ (S234d) ◁S225b S226b▷

`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.

S226b. ⟨*C types* S225b⟩+≡ (S234d) ◁S226a

`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
```

H.5 Prettyprinting C types

We have 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 definition, and the short method is used for everything else. The functions are mutually recursive, so they go into one big nest.

S226c. ⟨*prettyprinting C types* S226c⟩≡ (S234d) S227a▷

```
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 we're writing a field declaration, we want the code to look nice, so if the type ends in a star (i.e., it's a pointer type), we don't put a space between the type and the field name. That way we get declarations like "`Value v;`" and "`Exp *e;`", but never anything like "`Exp * e;`", which is ugly.

S227a. ⟨*prettyprinting C types* S226c⟩+≡ (S234d) ◁S226c S227b▷

```
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
```

In a long type declaration, we give 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.

S227b. ⟨*prettyprinting C types* S226c⟩+≡ (S234d) ◁S227a S227c▷

```
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. Function `foldr1` easily implements the C convention that an empty list of arguments is given by a prototype like `f(void)`.

S227c. ⟨*prettyprinting C types* S226c⟩+≡ (S234d) ◁S227b

```
type cons_proto = name * (ctype * name * field list)
protodoc : cons_proto -> doc

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
```

H.6 Creating C types from sums and products

Once we have a sum or product in the form of a `def`, we convert a sum to a tagged union, which means "`struct` containing `enum` and `union`," and we convert a product to a `struct`.

Because the `ctype` representation is set up to be easy to prettyprint, not to be easy to create, we proved convenience functions for creating `struct`, `union`, and pointer types.

S228a. ⟨*converting sums and products to C types* S228a⟩≡ (S234d) S228b ▷

```
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 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.

S228b. ⟨*converting sums and products to C types* S228a⟩+≡ (S234d) ◁ S228a S228c ▷

```
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.

- For a product, we get the fields from the arguments.
- For a sum, we have 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 will be `typedef`'d.

S228c. ⟨*converting sums and products to C types* S228a⟩+≡ (S234d) ◁ S228b S229a ▷

```
toCtype   : def -> ctype
mapOption : ('a -> 'b option) -> 'a list -> 'b list
```

```
⟨definitions of functions mapOption and camelCase S228d⟩
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 (mapOption altToFieldOption alts)
      in  withPtr (ptr, struct' (n, [FIELD (enum, altsuffix), FIELD (u, "")]))
      end
```

Function `mapOption f` applies `f` to a list of values and returns only the results that are not `NONE`.

S228d. ⟨*definitions of functions* `mapOption` *and* `camelCase` S228d⟩≡ (S228c) S229b ▷

```
mapOption : ('a -> 'b option) -> 'a list -> 'b list
```

```
fun mapOption f =
  let fun add (x, tail) = case f x of NONE => tail | SOME y => y :: tail
  in  foldr add []
  end
```

H.7 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: we also emit *constructor functions* for creating values of the types. Given a PRODUCT, we create a single constructor function. Given a SUM, we create a constructor function for each alternative in the sum. In both cases, when we create a function, we also create a prototype.

S229a. ⟨*converting sums and products to C types* S228a⟩+≡ (S234d) ◁S228c

```
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
```

To get the name of the constructor function, we start with mk, 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.

S229b. ⟨*definitions of functions* mapOption *and* camelCase S228d⟩+≡ (S228c) ◁S228d

```
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 is always complex. We begin with some auxiliary functions. Functions isPtr tells if a C type is a pointer type, and defSum tells if it is a sum.

S229c. ⟨*auxiliary functions for emitting a constructor function* S229c⟩≡ (S230d) S229d▷

```
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*. Normally the answer is called n, but if the name n conflicts with an argument, we keep adding more n's until we get a name that doesn't conflict. Value argfields is in scope and contains the fields that represent the arguments to the constructor function.

S229d. ⟨*auxiliary functions for emitting a constructor function* S229c⟩+≡ (S230d) ◁S229c S230a▷

```
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
```

We'd like to write code that manipulates the answer, but we don't know what the answer is going to be called. Function ans enables us to refer to the answer as % within a string.

S230a. ⟨*auxiliary functions for emitting a constructor function* S229c⟩+≡ (S230d) ◁S229d S230b▷

```
val ans =                                          ans : string -> doc
  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).

S230b. ⟨*auxiliary functions for emitting a constructor function* S229c⟩+≡ (S230d) ◁S230a S230c▷

```
                                            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.

S230c. ⟨*auxiliary functions for emitting a constructor function* S229c⟩+≡ (S230d) ◁S230b

```
                                            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, here is the prettyprinting document that represents the definition of a constructor function:

S230d. ⟨*functions that build documents to be emitted* S230d⟩≡ (S234d) S231a▷

```
                                    consFunDoc : def -> cons_proto -> doc
fun consFunDoc def (proto as (cname, (result, fname, argfields))) =
  let ⟨auxiliary functions for emitting a constructor function S229c⟩
  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
```

H.8 Writing the output

This program'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.)

S231a. ⟨*functions that build documents to be emitted* S230d⟩+≡ (S234d) ◁S230d S231b▷

```
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 we reuse `fieldDoc`.

S231b. ⟨*functions that build documents to be emitted* S230d⟩+≡ (S234d) ◁S231a S231c▷

`typedefdoc : typedef -> doc`

```
fun typedefdoc (ty, name) =
  agrp (doc "typedef " ^^ fieldDoc (FIELD (ty, name)) ^^ semi)
```

We emit a `typedef` for every definition, plus additional `typedef`s for internal, named types.

S231c. ⟨*functions that build documents to be emitted* S230d⟩+≡ (S234d) ◁S231b S231d▷

```
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
```

We emit definitions for every tagged type, which in practice includes only `struct` types.

S231d. ⟨*functions that build documents to be emitted* S230d⟩+≡ (S234d) ◁S231c S231e▷

```
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. For a function definition, we call `consFunDoc`. Function definitions are separated by blank lines.

S231e. ⟨*functions that build documents to be emitted* S230d⟩+≡ (S234d) ◁S231d

```
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
```

We write constructor functions to a C file, and we write definitions of four `noweb` chunks to a `.xnw` file.

S231f. ⟨*function* `process`*, which reads input and writes output* S231f⟩≡ (S234d)

```
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
```

H.9 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 (1999) prettyprinter. The definition of doc simply gives the alternatives.

S232a. ⟨*definition of* doc *and functions* S232a⟩≡ (S225a) S232b ▷

```
datatype doc
  = ^^     of doc * doc
  | TEXT   of string
  | BREAK  of string
  | INDENT of int * doc
  | GROUP  of break_line or_auto * doc
```

The grouping mechanisms 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:

S232b. ⟨*definition of* doc *and functions* S232a⟩+≡ (S225a) ◁ S232a S232c ▷

```
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
```

Because the ML constructors can be awkward to use, we provide convenience functions.

S232c. ⟨*definition of* doc *and functions* S232a⟩+≡ (S225a) ◁ S232b S233 ▷

```
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. It turns out to be easier to understand the code if we solve 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:

- If we remove a `d ^^ d'` from the head of the list, we put back `d` and `d'` separately.
- If we remove a `TEXT s` from the head of the list, we add `s` to the result list.
- If we remove an `INDENT (i, d)` from the head of the list, we replace it with `d`, appropriately tagged with the additional indentation.
- If we remove a `BREAK` from the head of the list, we may or may not add a newline and indentation to the result, depending on the break mode and the space available.
- If we remove a `GROUP(AUTO, d)` from the head of the list, we tag `d` with either `Flat` or `Break`, depending on space available, and we put it back on the head of the list.
- If we remove any other kind of `GROUP(B mode, d)` from the head of the list, we tag `d` with `mode` and put it 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 `doc`s. "Putting an item back on the head of the list" is accomplished with internal function `reformat`.

S233. ⟨*definition of* `doc` *and functions* S232a⟩+≡ (S225a) ◁S232c S234a▷

```
format : int -> int -> (int * break_line * doc) list -> string list
```

```
fun format w k [] = []
  | format w k (tagged_doc :: z) =
      let 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
          fun reformat item = format w k (item::z)
      in  case tagged_doc
            of (i,b, x ^^ y)        => format w k ((i,b,x)::(i,b,y)::z)
             | (i,b,TEXT s)         => addString s
             | (i,b,INDENT(j,x))    => reformat (i+j,b,x)
             | (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, x))  => if fits (w - k, (i,NO,x) :: z)
                                        then reformat (i,NO,x)
                                        else reformat (i,YES,x)
             | (i,b,GROUP(B break,x)) => reformat (i,break,x)
      end
```

[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.

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.

S234a. ⟨*definition of* `doc` *and functions* S232a⟩+≡ (S225a) ◁S233 S234b▷

```
and fits (w, []) = w >= 0    fits : int * (int * break_line * doc) list -> bool
  | 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 we reach 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 ensures that we 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 takes the problem of laying a single document and converts it to an instance of the more general problem: 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.

S234b. ⟨*definition of* `doc` *and functions* S232a⟩+≡ (S225a) ◁S234a

```
fun layout w doc = concat (format w 0 [(0, NO, GROUP (AUTO, doc))])
```

H.10 Putting everything together

S234c. ⟨*lexical analysis and parsing for μASDL* S234c⟩≡ (S234d)

⟨*lexical analysis for μASDL* S222a⟩
⟨*parsers for μASDL* S223a⟩

S234d. ⟨*asdl.sml* S234d⟩≡

⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* generated automatically⟩

⟨*abstract syntax for μASDL* S222d⟩

⟨*lexical analysis and parsing for μASDL* S234c⟩

⟨*prettyprinting combinators* S225a⟩
⟨*C types* S225b⟩
⟨*prettyprinting C types* S226c⟩
⟨*converting sums and products to C types* S228a⟩
⟨*functions that build documents to be emitted* S230d⟩
⟨*function* `process`*, which reads input and writes output* S231f⟩

```
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
```

Chapter contents

Code for writing interpreters in ML

I

Just as Appendix F presents reusable infrastructure for building interpreters in C, this appendix presents reusable infrastructure for building interpreters in ML. This code 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.

S237a. ⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S237a⟩≡ (S373a)

```
⟨for working with curried functions: id, fst, snd, pair, curry, and curry3 S263d⟩
⟨support for names and environments 310a⟩
⟨support for detecting and signaling errors detected at run time S366c⟩
⟨list functions not provided by Standard ML's initial basis S241b⟩
⟨utility functions for string manipulation and printing S238a⟩
⟨support for representing errors as ML values S243b⟩
⟨type interactivity plus related functions and value S368a⟩
⟨simple implementations of set operations S240b⟩
⟨collections with mapping and combining functions S240c⟩
⟨suspensions S249a⟩
⟨streams S250a⟩
⟨stream transformers and their combinators S261a⟩
⟨support for source-code locations and located streams S254d⟩
⟨streams that track line boundaries S272a⟩
⟨support for lexical analysis S268b⟩
⟨common parsing code S260⟩
⟨shared utility functions for initializing interpreters S372b⟩
⟨function application with overflow checking S242b⟩
⟨function forward, for mutual recursion through mutable reference cells S243a⟩
exception LeftAsExercise of string
```

All interpreters that include type checkers incorporate this code:

S237b. ⟨*exceptions used in languages with type checking* S237b⟩≡

```
exception TypeError of string
exception BugInTypeChecking of string
```

And all interpreters that implement type inference incorporate this code:

S237c. ⟨*exceptions used in languages with type inference* S237c⟩≡

```
exception TypeError of string
exception BugInTypeInference of string
```

I.1 Reusable utility functions

This section includes small utility functions for printing, for manipulating automatically generated names, and for manipulating sets.

I.1.1 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 writes a string followed by a newline. Functions `eprint` and `eprintln` are analogous to `print` and `println`, but they write to standard error. It would be nice to be able to define more sophisticated printing functions like the ones in Section 1.6.1 on page 46, but making such functions type-safe requires code that beginning ML programmers would find baffling.

S238a. ⟨*utility functions for string manipulation and printing* S238a⟩≡ (S237a) S238b ▷

```
fun println  s = (print s; print "\n")
fun eprint   s = TextIO.output (TextIO.stdErr, s)
fun eprintln s = (eprint s; eprint "\n")
```

CLOSING IN ON CHECK-PRINT:

S238b. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁ S238a S238c ▷

```
val xprinter = ref print
fun xprint   s = !xprinter s
fun xprintln s = (xprint s; xprint "\n")
```

S238c. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁ S238b S238d ▷

```
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
```

S238d. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁ S238c S238e ▷

```
fun bprinter () =
  let val buffer = ref []
      fun bprint s = buffer := s :: !buffer
      fun contents () = concat (rev (!buffer))
  in  (bprint, contents)
  end
```

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.

S238e. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁ S238d S238f ▷

```
fun predefinedFunctionError s = eprintln ("while reading predefined functions, " ^ s)
```

Standard ML's built-in support for converting integers to strings uses the ~ character as a minus sign. We want the hyphen.

S238f. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁ S238e S238g ▷

```
fun intString n =
  String.map (fn #"~" => #"-" | c => c) (Int.toString n)
```

`intString : int -> string`

Plurals!

S238g. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁ S238f S239a ▷

```
fun plural what [x] = what
  | plural what _   = what ^ "s"

fun countString xs what =
  intString (length xs) ^ " " ^ plural what xs
```

Lists! Functions spaceSep and commaSep are special cases of the more general function separate.

S239a. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁S238g S239b▷

```
spaceSep :                   string list -> string
commaSep :                   string list -> string
separate : string * string -> string list -> string
```

```
fun separate (zero, sep) =
  (* list with separator *)
  let fun s []     = zero
        | s [x]    = x
        | s (h::t) = h ^ sep ^ s t
  in  s
end
val spaceSep = separate ("", " ")   (* list separated by spaces *)
val commaSep = separate ("", ", ")  (* list separated by commas *)
```

Here's how we print Unicode characters.

S239b. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁S239a S239c▷

```
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
```

To hash strings, I use an algorithm by Glenn Fowler, Phong Vo, and Landon Curt Noll. The "offset basis" has been adjusted by removing the high bit, so the computation works using 31-bit integers. http://tools.ietf.org/html/draft-eastlake-fnv-03 http://www.isthe.com/chongo/tech/comp/fnv/

S239c. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁S239b S240a▷

```
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
```


RuntimeError S366c

I.1.2 Utility functions for renaming variables

In the theory of programming languages, it's fairly common to talk about *fresh names*, where "fresh" means "different from any name in the program or its environment." 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`.

S240a. ⟨*utility functions for string manipulation and printing* S238a⟩+≡ (S237a) ◁S239c

```
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
```

I.1.3 Utility functions for sets, collections, and lists

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]

S240b. ⟨*simple implementations of set operations* S240b⟩≡ (S237a)

```
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
```

```
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
```

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 (for an example, see Exercise 38 on page 530). Here is a type `collection` that is distinct from the set/list type.

S240c. ⟨*collections with mapping and combining functions* S240c⟩≡ (S237a) S241a▷

```
datatype 'a collection = C of 'a set
fun elemsC (C xs) = xs
fun singleC x      = C [x]
val emptyC         = C []
```

```
type 'a collection
elemsC  : 'a collection -> 'a set
singleC : 'a -> 'a collection
emptyC  :       'a collection
```

[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.

The really useful functions are below: together with `singleC`, functions `joinC` and `mapC` 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.)

Here are the key functions:

- 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 quite 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 resulting collection has size $N \times M$.

S241a. ⟨*collections with mapping and combining functions* S240c⟩+≡ (S237a) ◁S240c

```
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)
```

Sometimes we need to zip together three lists of equal length.

S241b. ⟨*list functions not provided by Standard ML's initial basis* S241b⟩≡ (S237a) S241c ▷

```
zip3   : 'a list * 'b list * 'c list -> ('a * 'b * 'c) list
unzip3 : ('a * 'b * 'c) list -> 'a list * 'b list * 'c list
```

```
fun zip3 ([], [], []) = []
  | zip3 (x::xs, y::ys, z::zs) = (x, y, z) :: zip3 (xs, ys, zs)
  | zip3 _ = raise ListPair.UnequalLengths

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
```

Standard ML's list-reversal function is called `rev`, but in this book I use `reverse`.

S241c. ⟨*list functions not provided by Standard ML's initial basis* S241b⟩+≡ (S237a) ◁S241b S242a ▷

```
val reverse = rev
```

S242a. ⟨*list functions not provided by Standard ML's initial basis* S241b⟩+≡ (S237a) ◁S241c

```
fun optionList [] = SOME []      optionList : 'a option list -> 'a list option
  | optionList (NONE :: _) = NONE
  | optionList (SOME x :: rest) =
      (case optionList rest
         of SOME xs => SOME (x :: xs)
          | NONE    => NONE)
```

I.1.4 Utility function for limiting the depth of recursion

If there's no other overhead, MLton delivers 25 million evals per second. Finding all solutions to a Boolean formula requires on the order of 200.

S242b. ⟨*function application with overflow checking* S242b⟩≡ (S237a)

```
local
  val throttleCPU = case OS.Process.getEnv "BPCOPTIONS"
                      of SOME "nothrottle" => false
                       | _ => true
  val defaultRecursionLimit = 6000 (* about 1/5 of 32,000? *)
  val recursionLimit = ref defaultRecursionLimit
  val evalFuel       = ref 1000000
  datatype checkpoint = RECURSION_LIMIT of int
in
  val defaultEvalFuel = ref (!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

  fun fuelRemaining () = !evalFuel

  fun checkpointLimit () = RECURSION_LIMIT (!recursionLimit)
  fun restoreLimit (RECURSION_LIMIT n) = recursionLimit := n

  fun applyCheckingOverflow f =
    if !recursionLimit <= 0 then
      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 resetOverflowCheck () = ( recursionLimit := defaultRecursionLimit
                              ; evalFuel := !defaultEvalFuel
                              )
end
```

I.1.5 Utility function for mutual recursion

In Standard ML, mutually recursive functions are typically defined using the `and` keyword. But such a definition requires that the functions be adjacent in the source code. When there are large mutual recursions in which many functions participate, it is often simpler to implement mutual recursion the way a C programmer does: put each function in a mutable reference cell and call indirectly through the contents of that cell. But how is the cell to be initialized? In C, initialization is handled by the linker. In ML, we have to initialize the reference cell when we create it; the cell doesn't get its final value until the function it refers to is defined. To initialize such a cell, I use function `forward` to create an initial function. That initial function, if ever called, causes a fatal error.

S243a. ⟨*function* `forward`*, for mutual recursion through mutable reference cells* S243a⟩≡ (S237a)

```
fun forward what _ =
  let exception UnresolvedForwardDeclaration of string
  in  raise UnresolvedForwardDeclaration what
  end
```

For an example of `forward`, see \chunkref: chunk.first-use-of-forward. (THIS COULD POSSIBLY BE ELIMINATED.)

I.2 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 make the control transfer work out the way it should. If you need to get the code right, sometimes it's 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:

  S243b. ⟨*support for representing errors as ML values* S243b⟩≡ (S237a) S244a ▷

  ```
  datatype 'a error = OK of 'a | ERROR of string
  ```

- My parsers, which use technology described in Appendix J 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.

  `RuntimeError` S366c

  The following transcript gives an example of each case:

  ```
  -> (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 to make sure every exception is caught. 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`? Standard function composition and the expression `g (f x)` don't exactly make sense, but the *idea* of composition is good. This form of composition poses a standard problem, and it has a standard solution. The solution 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 that we apply `f` to `x`, and if the result is `OK y`, we can 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 traditional for a continuation.

S244a. ⟨*support for representing errors as ML values* S243b⟩+≡ (S237a) ◁S243b S244b▷

```
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."

S244b. ⟨*support for representing errors as ML values* S243b⟩+≡ (S237a) ◁S244a S244c▷

```
infix 1 >>=+                         >>=+ : 'a error * ('a -> 'b) -> 'b error
fun e >>=+ k'  =  e >>= (OK o k')
```

Sometimes we 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. I call the conversion operation `errorList`.[4] I implement it by folding over the list of possibly erroneous results, concatenating *all* error messages.

S244c. ⟨*support for representing errors as ML values* S243b⟩+≡ (S237a) ◁S244b S245a▷

```
                                     errorList : 'a error list -> 'a list error
fun errorList es =
  let fun cons (OK x, OK xs) = OK (x :: xs)
```

[2] "Roughly" because in truth, the answer also includes unread input.

[3] Haskell uses `flip fmap`.

[4] Haskell calls it `sequence`.

```
          | 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
```

These functions are used in parsing and elsewhere.

S245a. ⟨*support for representing errors as ML values* S243b⟩+≡ (S237a) ◁S244c

```
fun errorLabel s (OK x) = OK x
  | errorLabel s (ERROR msg) = ERROR (s ^ msg)
```


I.3 Unit testing

When running a unit test, we have 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, we had to fool around with `set_error_mode`. In ML, things are simpler: we convert the result of evaluation either to `OK` v, where v is a value, or to `ERROR` m, where m is an error message, as described above. On top of this representation, I build some shared utility functions.

When a `check-expect` fails, function `whatWasExpected` reports what was expected. If the thing expected was a syntactic value, I show just the value. Otherwise I show the syntax, plus whatever the syntax evaluated to. The definition of `asSyntacticValue` is language-dependent.

S245b. ⟨*shared* `whatWasExpected` S245b⟩≡ (S246c)

```
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
```

Function `checkExpectPasses` runs a `check-expect` test and tells 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`.

S245c. ⟨*shared* `checkExpectPassesWith`*, which calls* `outcome` S245c⟩≡ (S246c)

```
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 ",
```

Unit-testing functions provided by each language	
`outcome`	`: exp -> value error`
`ty`	`: exp -> ty error`
`testEqual`	`: value * value -> bool`
`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`

Table I.1: Unit-testing functions

```
                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`.

S246a. ⟨*shared* `checkAssertPasses` *and* `checkErrorPasses`, *which call* `outcome` S246a⟩≡ (S246c) S246b ▷

```
val cafailed = "check-assert failed: "          checkAssertPasses : exp -> bool
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`.

S246b. ⟨*shared* `checkAssertPasses` *and* `checkErrorPasses`, *which call* `outcome` S246a⟩+≡ (S246c) ◁ S246a

```
val cefailed = "check-error failed: "          checkErrorPasses : exp -> bool
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]
```

S246c. ⟨*shared* `check{Expect,Assert,Error{Passes`, *which call* `outcome` S246c⟩≡ (S378a)

```
⟨shared whatWasExpected S245b⟩
⟨shared checkExpectPassesWith, which calls outcome S245c⟩
⟨shared checkAssertPasses and checkErrorPasses, which call outcome S246a⟩
fun checkExpectPasses (cx, ex) = checkExpectPassesWith testEqual (cx, ex)
```

Here is the promised `failtest`.

S246d. ⟨*shared unit-testing utilities* S246d⟩≡ (S369b) S247a ▷

```
fun failtest strings = (app eprint strings; eprint "\n"; false)
```

In each bridge language, test results are reported the same way. If there are no tests, there is no report. (The report's format is stolen from the DrRacket programming environment.)

S247a. ⟨*shared unit-testing utilities* S246d⟩+≡ (S369b) ◁ S246d

```
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.

S247b. ⟨*shared definition of* processTests S247b⟩≡ (S369b)

processTests : unit_test list * basis -> unit

```
fun numberOfGoodTests (tests, rho) =
  foldr (fn (t, n) => if testIsGood (t, rho) then n + 1 else n) 0 tests
fun processTests (tests, rho) =
      reportTestResults (numberOfGoodTests (tests, rho), length tests)
```

S247c. ⟨*global variables and exception for counting assertions* S247c⟩≡

```
exception AssertionFailure of srcloc * string
val assertionsPassed  = ref 0
val assertionsChecked = ref 0
```

S247d. ⟨*other handlers that catch non-fatal exceptions and pass messages to* caught **[[assertions]]** S247d

```
| AssertionFailure (loc, expstring) =>
    if !toplevel_error_format = WITHOUT_LOCATIONS andalso fst loc = "standard in
    then
      caught ("Assertion " ^ expstring ^ " failed")
    else
      caught ("Assertion " ^ expstring ^ " failed at " ^ srclocString loc)
```

S247e. ⟨*code that reports on assertions, just before exit* S247e⟩≡

```
val () = reportTestResultsOf "assertion" (!assertionsPassed, !assertionsChecked)
```

I.4 Polymorphic streams, with optional side effects

A parser defines a function from a sequence of input lines to a sequence of extended definitions. 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, from which we build getline_. In ML, the imperative library function is TextIO.inputLine. But in both languages, once you get the line, it's gone, and you can't get it again. But it is possible to choose another representation of sequences that turns a sequence

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
```

Table I.2: Stream-related types and functions of imperative operations into an actual sequence data structure. That data structure is called a *stream*. By hiding the action of reading behind the stream abstraction, we can treat an input as an immutable sequence of lines... or characters... or extended definitions. The stream puts ephemeral results of unrepeatable actions into a data structure that we can hold onto as long as we like and examine 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 I.2.

I.4.1 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`. The suspension is created by passing the 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 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 but simply 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.

S249a. ⟨*suspensions* S249a⟩≡ (S237a) S249b ▷

```
datatype 'a action
  = PENDING  of unit -> 'a
  | PRODUCED of 'a

type 'a susp = 'a action ref
```

```
type 'a susp
```

Functions `delay` and `demand` convert to and from suspensions.

S249b. ⟨*suspensions* S249a⟩+≡ (S237a) ◁ S249a

```
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
```

```
delay  : (unit -> 'a) -> 'a susp
demand : 'a susp -> 'a
```

I.4.2 Streams: results of a sequence of actions

An interpreter has to perform not just one action but a whole sequence. If the goal is to read definitions, then the low-level action on top of which other actions are built is "read a line of input." But an interactive interpreter doesn't just read all the input and then convert it all to definitions. Instead, it reads just as much input as is needed to make the first definition, then evaluates the definition and prints the result. To orchestrate all these actions, I use *streams*.

[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`.

A stream behaves much like a list, except that the first time we look at each element, some 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 a large collection of related utility functions. Many of the utility functions are directly inspired by list functions like `map`, `filter`, `concat`, `zip`, and `foldl`.

Stream representation and basic functions

My representation of streams uses three cases:[6]

- The `EOS` constructor represents an empty stream.
- The `:::` constructor (pronounced "cons"), which I intend 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 standard `::` constructor, the `:::` constructor is written as an infix operator.
- The `SUSPENDED` constructor represents a stream in which the action need 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.

S250a. ⟨*streams* S250a⟩≡ (S237a) S250b ▷

```
datatype 'a stream
  = EOS
  | :::       of 'a * 'a stream
  | SUSPENDED of 'a stream susp
infixr 3 :::
```

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`).

S250b. ⟨*streams* S250a⟩+≡ (S237a) ◁ S250a S250c ▷

`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` constructors. It can also be convenient to create a stream from a list. When such a stream is read, no new actions are performed.

S250c. ⟨*streams* S250a⟩+≡ (S237a) ◁ S250b S250d ▷

`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.

S250d. ⟨*streams* S250a⟩+≡ (S237a) ◁ S250c S251a ▷

`listOfStream : 'a stream -> 'a list`

```
fun listOfStream xs =
  case streamGet xs
    of NONE => []
     | SOME (x, xs) => x :: listOfStream xs
```

[6] There are representations that use fewer cases, but this one has the merit that I can define a polymorphic empty stream without running afoul of ML's "value restriction."

[7] To help with debugging, I sometimes violate the abstraction and look at the state of a `SUSPENDED` stream.

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.

S251a. ⟨*streams* S250a⟩+≡ (S237a) ◁S250d S251b▷

```
fun delayedStream action =
  SUSPENDED (delay action)
```

`delayedStream : (unit -> 'a stream) -> 'a stream`

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`.

S251b. ⟨*streams* S250a⟩+≡ (S237a) ◁S251a S251c▷

`streamOfEffects : (unit -> 'a option) -> 'a stream`

```
fun streamOfEffects action =
  delayedStream (fn () => case action () of NONE   => EOS
                                          | SOME a => a ::: streamOfEffects action)
```

I use `streamOfEffects` to produce a stream of lines from an input file:

S251c. ⟨*streams* S250a⟩+≡ (S237a) ◁S251b S251d▷

```
type line = string
fun filelines infile =
  streamOfEffects (fn () => TextIO.inputLine infile)
```

```
type line
filelines : TextIO.instream -> line stream
```

Where `streamOfEffects` produces the results of repeating a single *action* again and again, `streamRepeat` simply 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 of any type:

S251d. ⟨*streams* S250a⟩+≡ (S237a) ◁S251c S251e▷

```
fun streamRepeat x =
  delayedStream (fn () => x ::: streamRepeat x)
```

`streamRepeat : 'a -> 'a stream`

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. By repeatedly applying the function, we 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*.

S251e. ⟨*streams* S250a⟩+≡ (S237a) ◁S251d S252a▷

`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')
```

delay	S249b
demand	S249b

Function `streamOfUnfold` can turn any "get" function into a stream. In fact, the standard 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.

S252a. ⟨*streams* S250a⟩+≡ (S237a) ◁S251e S252b▷

```
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. Function `streamFold` is defined below in chunk S253b.)

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 called `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.)

S252b. ⟨*streams* S250a⟩+≡ (S237a) ◁S252a S252c▷

```
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.

S252c. ⟨*streams* S250a⟩+≡ (S237a) ◁S252b S252d▷

```
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.

S252d. ⟨*streams* S250a⟩+≡ (S237a) ◁S252c S253a▷

```
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)
```

S253a. ⟨*streams* S250a⟩+≡ (S237a) ◁S252d S253b▷

```
fun streamFilter p xs =        streamFilter : ('a -> bool) -> 'a stream -> 'a stream
  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.

S253b. ⟨*streams* S250a⟩+≡ (S237a) ◁S253a S253c▷

```
fun streamFold f z xs =        streamFold : ('a * 'b -> 'b) -> 'b -> 'a stream -> 'b
  case streamGet xs of NONE => z
                     | SOME (x, xs) => streamFold f (f (x, z)) xs
```

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.

S253c. ⟨*streams* S250a⟩+≡ (S237a) ◁S253b S253d▷

```
fun streamZip (xs, ys) =       streamZip : 'a stream * 'b stream -> ('a * 'b) stream
  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 A's into a single stream of A'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)`.

S253d. ⟨*streams* S250a⟩+≡ (S237a) ◁S253c S253e▷

```
fun streamConcat xss =         streamConcat : 'a stream stream -> 'a stream
  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
```

The composition of `concat` with `map f` is very common in list and stream processing, so I give it a name.

`:::`	S250a
`delayedStream`	S251a
`EOS`	S250a
`streamGet`	S250b
`streamOfList`	S250c
`streamOfUnfold`	S251e

S253e. ⟨*streams* S250a⟩+≡ (S237a) ◁S253d S253f▷

```
                      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 simply implement append using concatenation.

S253f. ⟨*streams* S250a⟩+≡ (S237a) ◁S253e S254a▷

```
infix 5 @@@                               @@@ : 'a stream * 'a stream -> 'a stream
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.

S254a. ⟨*streams* S250a⟩+≡ (S237a) ◁S253f S254b▷

```
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 => []
```


If I want "take," sooner or later I'm sure to want "drop" (chunk S256b).

S254b. ⟨*streams* S250a⟩+≡ (S237a) ◁S254a

```
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
```

I.4.3 *Streams of extended definitions*

Every language has its own parser, called `xdefstream`, which converts a stream of lines to a stream of `xdefs`. But as in Section F.1.3, the convenience functions `filexdefs` and `stringsxdefs` are shared.

S254c. ⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S254c⟩≡ (S373b)

```
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)
```

I.5 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, the desire for precision has to be balanced against the need for simplicity. The best compromise is to track only source file and line number. That's good enough to help programmers find errors, and it eliminates bookkeeping that would otherwise be needed to track column numbers.

S254d. ⟨*support for source-code locations and located streams* S254d⟩≡ (S237a) S254e▷

```
type srcloc
srclocString : srcloc -> string
```

```
type srcloc = string * int
fun srclocString (source, line) =
  source ^ ", line " ^ intString line
```

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.4.1 on page S193, I use an *error format* to control when error messages include source-code locations. The format is initially set to include them.

S254e. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S254d S255a▷

```
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.

S255a. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S254e S255b▷

```
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)
```


Source locations are also used at run time. Any exception can be marked with a location by converting it to the Located exception:

S255b. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S255a S255c▷

```
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."

S255c. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S255b S255d▷

```
type 'a located = srcloc * 'a
```

`type 'a located`

To raise the Located exception, we use function atLoc. Calling atLoc f x applies f to x within the scope of handlers that convert recognized exceptions to the Located exception:

S255d. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S255c S255e▷

`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 S255f⟩
```

And we can call atLoc easily by using the higher-order function located:

S255e. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S255d S255g▷

`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)
```

Here are handlers for more exceptions we recognize. These handlers can be augmented by other, language-specific handlers.

S255f. ⟨*more handlers for* atLoc S255f⟩≡ (S255d)

```
| 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)
```

Once we have a location, we use it to fill in a template for an error message. The location replaces the string "<at loc>". The necessary string processing is done by fillComplaintTemplate, which relies on Standard ML's Substring module.

S255g. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S255e S256a▷

`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 an error at a given location, code calls `errorAt`.

S256a. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S255g S256b▷

```
  fun errorAt msg loc =                      errorAt : string -> srcloc -> 'a error
    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.

S256b. ⟨*support for source-code locations and located streams* S254d⟩+≡ (S237a) ◁S256a

```
                  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
```

I.6 Further reading

The `'a error` abstraction is an old functional-programming trick, first described by Spivey (1990). Ramsey (1999) demonstrates the use of this abstraction to suppress error messages in compilers.

ERROR	S243b
naturals	S252a
streamDrop	S254b
streamRepeat	S251d
streamZip	S253c
synerrormsg	S255a

Chapter contents

J

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 unless they have access to a special-purpose language-based editor (perhaps as part of an integrated development environment), programmers have to specify an abstract-syntax tree indirectly, by writing a sequence of characters. The process of turning a sequence of characters into syntax is called *parsing*.

Wait! It gets better. Quite often characters are turned into syntax in *two* stages: first characters are grouped together into *tokens*. Then, a parser turns a sequence of tokens into syntax. Think of a token as a word or a symbol or a punctuation mark.

Parsing is a deep, broad, well-developed topic with many interesting intellectual byways. A 500-page monograph on parsing was already famous in the 1970s, and clever minds have invented plenty of new techniques since then. Many techniques rely on a separate tool called a *parser generator*. The technique I use in this book requires no separate tools: I use *hand-written, recursive-descent parsers*. To help me 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:

- A parser might cooperate with the I/O routines to arrange that a suitable *prompt* is issued before each line is read. 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.
- If a parser encounters an error, it can't just give up. It needs 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 I.4.2. 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.

[1] I say "created," but a more accurate term would be "stolen."

- 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 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 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 the ultimate goal of a lexer is to produce a stream of tokens.

- 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 ultimate goal of a lexer is to produce a stream of abstract-syntax trees.

The polymorphic functions are described in Table J.1 on page S262; the specialized functions are described in Table J.2 on page S269.

The code is divided among these chunks:

S260. ⟨*common parsing code* S260⟩≡
  ⟨*combinators and utilities for parsing located streams* S272c⟩
  ⟨*transformers for interchangeable brackets* S274⟩
  ⟨*code used to debug parsers* S277d⟩
  ⟨*streams that issue two forms of prompts* S279a⟩

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.

J.1 Stream transformers, which act as parsers

Our ultimate goal is to turn streams of input lines into streams of definitions. Along the way we may also have 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.

S261a. ⟨*stream transformers and their combinators* S261a⟩≡ S261b ▷

```
type ('a, 'b) xformer =
  'a stream -> ('b error * 'a stream) option
```

`type ('a, 'b) xformer`

If we apply `streamOfUnfold`, from Section I.4.2, to an `('a, 'b) xformer`, we get 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.

J.1.1 Error-free transformers and their composition

The `pure` combinator takes a value `h` of type B as argument. It returns an A-to-B transformer that consumes no A's as input and produces `y`.

S261b. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁ S261a S263a ▷

```
fun pure y = fn xs => SOME (OK y, xs)
```

`pure : 'b -> ('a, 'b) xformer`

To build a stream transformer that reads inputs in sequence, we compose smaller stream transformers that read parts of the input. The sequential composition operator, if you have not seen it before, may look quite strange. To compose `tx_f` and `tx_b` in sequence, you use the infix operator `<*>`, which is pronounced "applied to." The composition is written `tx_f <*> tx_b`, and here's how it works:

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` which is a 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.

`type error`	S243b
`OK`	S243b
`type stream`	S250a

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. 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.

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
```

Table J.1: Stream transformers and their combinators

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.

S263a. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S261b S263b▷

```
<*> : ('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)
```


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.

S263b. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S263a S263c▷

```
<$> : ('b -> 'c) * ('a, 'b) xformer -> ('a, 'c) xformer
```

```
infixr 4 <$>
fun f <$> p = pure f <*> p
```

NEW!

S263c. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S263b S264a▷

```
infixr 3 <~>
fun f <~> a = curry fst <$> f <*> a
```

There are a variety of ways to create useful functions in the f position. Many such functions are Curried. Here are some of them.

S263d. ⟨*for working with curried functions:* id, fst, snd, pair, curry, *and* curry3 S263d⟩≡

```
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 name parses a name and exp parses an expression then in a let binding we can parse a name * exp pair by

```
pair <$> name <*> exp
```

(To parse μScheme, we would need also to parse the surrounding parentheses.) As another example, if in μScheme we have seen the keyword if, we can follow it by the parser

```
curry3 IFX <$> exp <*> exp <*> exp
```

which creates the syntax 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. To make a choice, as in "val or expression or define or use," we use 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, succeeding if either succeeds, detecting an error if either detects an

ERROR S243b
OK S243b
pure S261b

error, and failing only if both fail. To assure that the result has a predictable type no matter which transformer is used, both `t1` and `t2` have to have the same type.

S264a. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁ S263c S264b ▷

```
<|> : ('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 with a fold operator, but I need a "zero" parser that always fails.

S264b. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁ S264a S264c ▷

```
fun pzero _ = NONE
pzero : ('a, 'b) xformer
```

Because building choices from lists is common, I implement this special case as `anyParser`.

S264c. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁ S264b S264d ▷

```
anyParser : ('a, 'b) xformer list -> ('a, 'b) xformer
fun anyParser ts =
  foldr op <|> pzero ts
```

J.1.2 Ignoring results produced by transformers

If a parser sees the stream of tokens

`(` `if` `(` `<` `x` `y` `)` `x` `y` `)`, ■

we 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 we do need to read them from the input, but we don'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 we don't care about. 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`.

S264d. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁ S264c S265a ▷

```
<*  : ('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
```

J.1.3 At last, transformers that look at the input stream

None of the transformers above looks directly at 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. It's time to meet those parsers.

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 input, `one` fails; it never signals an error.

S265a. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S264d S265b▷

```
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 we have reached the end of a stream. This parser, which is called `eos`, can produce no useful result, so it produces the empty tuple, which has type `unit`.

S265b. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S265a S265c▷

```
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 look at their input. The only other parsing combinator that looks directly at input is `stripAndReportErrors`, which removes `ERROR` and `OK` from error streams.

It is sometimes useful to look at input without consuming it. I provide two functions: `peek` just looks at a transformed stream and maybe produces a value, whereas `rewind` can change any transformer into a transformer that behaves identically, but 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.

S265c. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S265b S265d▷

```
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 we ran `tx`. The actions performed by `tx` can't be undone, but the inputs can be read again.

S265d. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S265c S266a▷

```
rewind : ('a, 'b) xformer -> ('a, 'b) xformer
```

```
fun rewind tx xs =
  case tx xs of SOME (ey, _) => SOME (ey, xs)
              | NONE => NONE
```


`<$>`	S263b
`<*>`	S263a
`curry`	S263d
`EOS`	S250a
`fst`	S263d
`OK`	S243b
`snd`	S263d
`streamGet`	S250b

J.1.4 Parsing combinators

Real parsers largely build on `<$>`, `<*>`, `<|>`, and `one` by adding the following ideas:

- Perhaps we'd like to succeed only if an input satisfies certain conditions. For example, if we're trying to read a number, we might want to write 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.

- Although I've avoided using "optional" syntax in my own designs, 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.

This section presents standard parsing combinators that help implement conditional parsers, parsers for sequences, and parsers for optional syntax.

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.

S266a. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S265d S266b▷

```
fun sat p tx xs =                      sat : ('b -> bool) -> ('a, 'b) xformer -> ('a, 'b) xformer
  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.

S266b. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S266a S266c▷

```
fun eqx y =                            eqx : ''b -> ('a, ''b) xformer -> ('a, ''b) xformer
  sat (fn y' => y = y')
```

A more subtle condition is that a partial function can turn an input into something we're looking for. If we have an A-to-B transformer, and we compose it with a function that given a B, sometimes produces a C, then we get an A-to-C transformer. Because there's a close analogy with the application operator `<$>`, I notate this *partial* application operator as `<$>?`, with a question mark.

S266c. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S266b S266d▷

```
infixr 4 <$>?                          <$>? : ('b -> 'c option) * ('a, 'b) xformer -> ('a, 'c) xformer
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)
```

We can run a parser conditional on the success of another parser. Parser `t1 <&> t2` succeeds only if both `t1` and `t2` succeed at the same point. This parser looks at enough input to decide if `t1` succeeds, but it does not consume that input—it consumes only the input of `t2`.

S266d. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S266c S267a▷

```
                                       <&> : ('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
```

We can also use the success or failure of a parser as a condition. Parser `notFollowedBy t` succeeds if and only if `t` fails. Parser `notFollowedBy t` may *look* at the input, but it never *consumes* any input. I use `notFollowedBy` when reading integer literals, to make sure that the digits are not followed by a letter or other non-delimiting symbol.

S267a. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S266d S267b▷

```
fun notFollowedBy t xs =        notFollowedBy : ('a, 'b) xformer -> ('a, unit) xformer
  case t xs
    of NONE => SOME (OK (), xs)
     | SOME _ => NONE
```

We now have something that resembles a little Boolean algebra for parsers: functions `<&>`, `<|>`, and `notFollowedBy` play the roles of "and," "or," and "not."

Parsers for sequences

Inputs are 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.

S267b. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S267a S267c▷

```
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. If you write your own parsers, it's a problem to watch out for.

Sometimes an empty list isn't acceptable. In that case, use `many1 t`, which succeeds only if `t` succeeds at least once—in which case it returns a nonempty list.

S267c. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S267b S267d▷

```
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. As it stands, if `t` can succeed without consuming any input, then `many t` does not terminate, so it is an unchecked run-time error to pass `many` a transformer that succeeds without consuming input. 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 S273c below.

Sometimes instead of zero, one, or many B's, we just one 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.

S267d. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S267c S268a▷

```
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.

`<$>`	S263b
`<*>`	S263a
`<|>`	S264a
`curry`	S263d
`ERROR`	S243b
`OK`	S243b
`pure`	S261b

J.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 duplicate name, an error should be signalled. We would transform the input into a value of type `name list error`. But the transformer type already includes the possibility of error, and we would prefer that errors detected by functions be on the same footing as errors detected by parsers, and that they be handled by the same mechanisms. To enable such handling, I define `<*>!` and `<$>!` combinators that merge function-detected errors with parser-detected errors.

S268a. ⟨*stream transformers and their combinators* S261a⟩+≡ ◁S267d

```
<*>! : ('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
```

J.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 transformers that are specialized to transform streams of characters into something else, usually tokens.

S268b. ⟨*support for lexical analysis* S268b⟩≡ S268c▷

```
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 parentheses. The other delimiter is the semicolon, which introduces a comment.

S268c. ⟨*support for lexical analysis* S268b⟩+≡ ◁S268b S270a▷

```
fun isDelim c =
  Char.isSpace c orelse Char.contains "()[]{};" c
```

`isDelim : char -> bool`

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
isLiteral              : string -> token -> bool
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
```

Table J.2: Transformers specialized for lexical analysis or parsing

<*>	S263a
ERROR	S243b
OK	S243b
pure	S261b

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.

S270a. ⟨*support for lexical analysis* S268b⟩+≡ ◁ S268c S270b ▷

```
val whitespace = many (sat Char.isSpace one)    whitespace : char list lexer
```

Most languages in this book are, like Scheme, very 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 doesn't form a name; it forms an integer literal. Because integer literals offer 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.

The rules for integer literals are as follows:

- The 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.

S270b. ⟨*support for lexical analysis* S268b⟩+≡ ◁ S270a S270c ▷

```
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 we 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.

S270c. ⟨*support for lexical analysis* S268b⟩+≡ ◁ S270b S270d ▷

```
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.

S270d. ⟨*support for lexical analysis* S268b⟩+≡ ◁ S270c S271a ▷

```
fun intToken isDelim =                          intToken : (char -> bool) -> int lexer
  intFromChars <$>! intChars isDelim
```

S271a. ⟨*support for lexical analysis* S268b⟩+≡ ◁S270d S271b▷

```
datatype bracket_shape = ROUND | SQUARE | CURLY

fun leftString ROUND  = "("
  | leftString SQUARE = "["
  | leftString CURLY  = "{"
fun rightString ROUND  = ")"
  | rightString SQUARE = "]"
  | rightString CURLY = "}"
```

Given a lexer for language tokens, we can build a lexer for tokens:

S271b. ⟨*support for lexical analysis* S268b⟩+≡ ◁S271a

```
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

fun plusBracketsString _   (LEFT shape)  = leftString shape
  | plusBracketsString _   (RIGHT shape) = rightString shape
  | plusBracketsString pts (PRETOKEN pt)  = pts pt
```

J.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 Section O.3.1 on page S373.

I hope transforming a stream of characters to a stream of tokens to a stream of definitions sounds appealing—but it simplifies the story a little too much. If nothing ever went wrong, it would be fine if all we ever saw were tokens. But if something does go wrong, I want to be able to do more than throw up my hands:

- I want say *where* things went wrong—at what *source-code location*.
- I want to get rid of the bad tokens that caused the error.
- I want to be able to start parsing over again interactively, without having to kill an interpreter and start over.

To support error reporting and recovery takes a lot of machinery.

J.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 bring its error processing to a prompt conclusion and ready itself to read the next line. To do so, it needs to know where the line boundaries are! For example, if I find an error on line 6, I want to read all the tokens on line 6, throw them away, and start over again on line 7. The nasty bit is that I want to do it *without* reading line 7—reading line 7 will take an action and will likely have the side effect of printing a prompt. And I want it to be the correct prompt. I therefore 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.[3]

S272a. ⟨*streams that track line boundaries* S272a⟩≡ S272b ▷

```
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
```

S272b. ⟨*streams that track line boundaries* S272a⟩+≡ ◁ S272a

```
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, we can now define parsers.

J.3.2 Parsing located, in-line tokens

A value of type `'a parser` takes a stream of located tokens set between end-of-line markers, and it returns a value of type `'a`, plus any leftover tokens.

S272c. ⟨*combinators and utilities for parsing located streams* S272c⟩≡ (S260) S273a ▷

```
type ('t, 'a) polyparser = ('t located eol_marked, 'a) xformer
```

[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. We don't.

[3]At some future point I may need to change `drainLine` to keep the `EOL` in order to track locations in μProlog.

The EOL and INLINE constructors are essential for error recovery, but for parsing, they just get in the way. Our first order of business is 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.

S273a. ⟨*combinators and utilities for parsing located streams* S272c⟩+≡ (S260) ◁S272c S273b▷

```
token    : ('t, 't)    polyparser
noTokens : ('t, unit) polyparser
```

```
fun token    stream = (snd <$> inline)      stream
fun noTokens stream = (notFollowedBy token) stream
```

Sometimes the easiest way to keep track of source-code locations is to pair a source-code location with a result from a parser. This happens just often enough that I find it worth while to 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.

S273b. ⟨*combinators and utilities for parsing located streams* S272c⟩+≡ (S260) ◁S273a S273c▷

```
@@ : ('t, 'a) polyparser -> ('t, 'a located) polyparser
```

```
fun @@ p = pair <$> srcloc <*> p
```

J.3.3 Parsers that report errors

Most syntactic forms (expressions, unit tests, definitions, and so on) are parsed by trying a set of alternatives. When all alternatives fail, I usually want to convert the failure into an error. 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. You can see an example in the parser for extended definitions in chunk S377e.

S273c. ⟨*combinators and utilities for parsing located streams* S272c⟩+≡ (S260) ◁S273b S273d▷

```
<?> : ('t, 'a) polyparser * string -> ('t, 'a) polyparser
```

```
infix 0 <?>
fun p <?> what = p <|> errorAt ("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, if we're 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 errorAt and srcloc in the same way that <?> does, because I have two goals: *consume* the tokens recognized by p, and also *report* the error at the location of the first of those tokens. I can't use errorAt 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.

S273d. ⟨*combinators and utilities for parsing located streams* S272c⟩+≡ (S260) ◁S273c S277c▷

```
<!> : ('t, 'a) polyparser * string -> ('t, 'b) polyparser
```

```
infix 4 <!>
fun p <!> msg =
```

```
fn tokens => (case p tokens
                of SOME (OK _, unread) =>
                     (case peek srcloc tokens
                        of SOME loc => SOME (errorAt msg loc, unread)
                         | NONE => NONE)
                 | _ => NONE)
```


J.3.4 Parsers for common programming-language idioms

This section defines special-purpose parsers and combinators which handle phrases and idioms that appear in many of the languages in this book.

Interchangeable 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. The `bracketKeyword`[4] function creates a parser that recognizes inputs of the form

(*keyword stuff*)

The `bracketKeyword` function embodies some useful error handling:

- It takes an extra parameter `expected`, which says, when anything goes wrong, what the parser was expecting in the way of *stuff*.
- If something does go wrong parsing *stuff*, it calls `scanToClose` to scan past all the tokens where *stuff* was expected, up to and including the matching close parenthesis. Function `scanToClose` returns `SOME` applied to the location where *stuff* was expected, or if there was no closing bracket, it returns `NONE`.

Once the parser sees the opening parenthesis and the keyword, failure is impossible: either parser `p` parses *stuff* correctly, or there's an error.

S274. ⟨*transformers for interchangeable brackets* S274⟩≡ (S260) S276a ▷

```
fun notCurly (_, CURLY) = false
  | notCurly _          = true

(* left: takes shape, succeeds or fails
   right: takes shape and
      succeeds with right bracket of correct shape
      errors with right bracket of incorrect shape
      fails with token that is not right bracket *)

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 leftCurly tokens = sat (not o notCurly) left tokens

fun atRight expected = rewind right <?> expected

fun badRight msg =
  (fn (loc, shape) => errorAt (msg ^ " " ^ rightString shape) loc) <$>! right
```

[4] 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. So the function is called `bracketKeyword`.

Parser `right` matches a right bracket by itself. But quite commonly, we want to wrap another parser `p` in matching left and right brackets. If something goes wrong—say the brackets don't match—we ought not to try to address the error in the right-bracket parser alone; we need to be able to report the location of the left bracket as well. To be able to issue good error messages, I define parser `matchingRight`, which always succeeds and which produces one of three outcomes:

- Result `FOUND_RIGHT` (loc, s) says we found a right bracket exactly where we expected to, and its shape and location are s and loc.
- Result `SCANNED_TO_RIGHT` loc says we didn't find a right bracket at loc, but we scanned to a matching right bracket eventually.
- Result `NO_RIGHT` says that we scanned the entire input without finding a matching right bracket.

Function `matchBrackets` takes this result, along with the left bracket and the parsed result a, and knows what to do.

S276a. ⟨*transformers for interchangeable brackets* S274⟩+≡ (S260) ◁S274 S276b▷

```
type right_result
matchingRight : ('t, right_result) pb_parser
scanToClose   : ('t, right_result) pb_parser
matchBrackets : string -> bracket_shape located -> 'a -> right_result -> 'a error
```

```
type ('t, 'a) pb_parser = ('t plus_brackets, 'a) polyparser
datatype right_result
  = FOUND_RIGHT      of bracket_shape located
  | SCANNED_TO_RIGHT of srcloc  (* location where scanning started *)
  | NO_RIGHT

fun scanToClose tokens =
  let val loc = getOpt (peek srcloc tokens, ("end of stream", 9999))
      fun scan lpcount tokens =
        (* lpcount is the number of unmatched left parentheses *)
        case tokens
          of EOL _                  ::: tokens => scan lpcount tokens
           | INLINE (_, LEFT  t)    ::: tokens => scan (lpcount+1) tokens
           | INLINE (_, RIGHT t)    ::: tokens => if lpcount = 0 then
                                                    pure (SCANNED_TO_RIGHT loc) tokens
                                                  else
                                                    scan (lpcount-1) tokens
           | INLINE (_, PRETOKEN _) ::: tokens => scan lpcount tokens
           | EOS         => pure NO_RIGHT tokens
           | SUSPENDED s => scan lpcount (demand s)
  in  scan 0 tokens
  end

fun matchingRight tokens = (FOUND_RIGHT <$> right <|> scanToClose) tokens

fun matchBrackets _ (loc, left) _ NO_RIGHT =
      errorAt ("unmatched " ^ leftString left) loc
  | matchBrackets e (loc, left) _ (SCANNED_TO_RIGHT loc') =
      errorAt ("expected " ^ e) loc
  | matchBrackets _ (loc, left) a (FOUND_RIGHT (loc', right)) =
      if left = right then
        OK a
      else
        errorAt (rightString right ^ " does not match " ^ leftString left ^
                 (if loc <> loc' then " at " ^ srclocString loc else "")) loc'
```

Story:

- Parser can fail, right bracket has to match: `liberalBracket`
- Keyword can fail, but if it matches, parser has to match: `bracketKeyword`
- Left bracket can fail, but if it matches, parser has to match: `bracket`, `curlyBracket`

S276b. ⟨*transformers for interchangeable brackets* S274⟩+≡ (S260) ◁S276a S277a▷

```
bracketKeyword : ('t, 'keyword) pb_parser * string * ('t, 'a) pb_parser -> ('t, 'a) p
```

```
fun liberalBracket (expected, p) =
  matchBrackets expected <$> sat notCurly left <*> p <*>! matchingRight
fun bracketKeyword (keyword, expected, p) =
  liberalBracket (expected, keyword *> (p <?> expected))
```

```
fun bracket (expected, p) =
  liberalBracket (expected, p <?> expected)
fun curlyBracket (expected, p) =
  matchBrackets expected <$> leftCurly <*> (p <?> expected) <*>! matchingRight
```

Usually, we want to pull the keyword out of the usage string.

S277a. ⟨*transformers for interchangeable brackets* S274⟩+≡ (S260) ◁S276b S277b▷

```
usageParser : (string -> ('t, string) pb_parser) -> string * ('t, 'a) pb_parser -> ('t, 'a) pb

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)
           | _ => let exception BadUsage of string in raise BadUsage usage end
  end
```

Hello, stranger?

S277b. ⟨*transformers for interchangeable brackets* S274⟩+≡ (S260) ◁S277a

```
fun pretoken stream = ((fn PRETOKEN t => SOME t | _ => NONE) <$>? token) stream
```

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.

S277c. ⟨*combinators and utilities for parsing located streams* S272c⟩+≡ (S260) ◁S273d

```
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 : string => y = x) xs then
                            errorAt (what ^ " " ^ x ^ " appears twice in " ^ conte
                          else
                            dup xs
  in  dup names
  end
```

Function List.exists is like the μScheme exists?. It is in the initial basis for Standard ML.

J.3.5 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 all the tokens that are available *without* triggering the action of reading another line of input. I believe it's safe to read until I have got to *both* an end-of-line marker *and* a suspension whose value has not yet been demanded.

S277d. ⟨*code used to debug parsers* S277d⟩≡ (S260) S278a▷

```
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
```

The showErrorInput function 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.

S278a. ⟨*code used to debug parsers* S277d⟩+≡ (S260) ◁S277d S278b▷

```
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
```

The wrapAround function can be used to wrap a parser; it shows what the parser was looking for, what tokens it was looking at, and whether it found something.

S278b. ⟨*code used to debug parsers* S277d⟩+≡ (S260) ◁S278a

```
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)
```

J.4 Streams that lex, parse, and prompt

In this final section I pull together all the machinery needed to take a stream of input lines, a lexer, and a parser, and to produce a stream of high-level syntactic objects like definitions. With prompts! This code is where prompts get determined, where errors are handled, and where special tagged lines are copied to the output to support testing.

Testing support

Let's get the testing support out of the way first. As in the C code, I want to print out any line read that begins with the special string ;#. This string is a formal comment that helps me test chunks marked ⟨*transcript*⟩. In the ML code, I can do the job in a very modular way: I define a post-stream action that prints any line meeting the criterion. Function `echoTagStream` transforms a stream of lines to a stream of lines, adding the behavior I want.

S279a. ⟨*streams that issue two forms of prompts* S279a⟩≡ (S260) S279b ▷

```
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

Function `stripAndReportErrors` removes the `ERROR` and `OK` tags from a stream, producing an output stream with a simpler type. Values tagged with `OK` are passed on to the output stream unchanged; messages tagged with `ERROR` are printed to standard error, using `eprintln`.

S279b. ⟨*streams that issue two forms of prompts* S279a⟩+≡ (S260) ◁ S279a S279c ▷

```
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
```

An error detected during lexical analysis is 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 I don't keep track of the line's source-code location.

S279c. ⟨*streams that issue two forms of prompts* S279a⟩+≡ (S260) ◁ S279b S279d ▷

```
lexLineWith : 't lexer -> line -> 't stream
```

```
fun lexLineWith lexer =
  stripAndReportErrors o streamOfUnfold lexer o streamOfList o explode
```

When an error occurs during parsing, I drain the rest of the tokens on the line where the error occurred. I *don't* strip the errors at this point; errors are passed on to the interactive stream because when an error is detected, the prompt may need to be changed.

S279d. ⟨*streams that issue two forms of prompts* S279a⟩+≡ (S260) ◁ S279c S280a ▷

```
parseWithErrors : ('t, 'a) polyparser -> 't located eol_marked 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`	S272a
`eprint`	S238a
`eprintln`	S238a
`ERROR`	S243b
`OK`	S243b
`postStream`	S252c
`safeTokens`	S277d
`spaceSep`	S239a
`streamGet`	S250b
`streamOfList`	S250c
`streamOfUnfold`	S251e

Prompts

All interpreters in the book are built on the Unix shell model of having 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. To turn prompting off, we set both to the empty string.

S280a. ⟨*streams that issue two forms of prompts* S279a⟩+≡ (S260) ◁S279d S280b▷

```
type prompts
stdPrompts : prompts
noPrompts  : prompts
```

```
type prompts   = { ps1 : string, ps2 : string }
val stdPrompts = { ps1 = "-> ", ps2 = "   " }
val noPrompts  = { ps1 = "", ps2 = "" }
```

Building a reader

Our last stream function does two jobs which are interconnected: it 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. We start with 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. Function `lexLineWith lexer` converts a line to a stream of tokens, which then are paired with source-code locations, tagged with `INLINE`, and followed by an `EOL` value. This extra decoration gets us from 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. The 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.

To deliver the right prompt in the right situation, I store 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 the prompt to `ps2`. But when we are about to get a new definition, 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, makes sure the prompt is always appropriate to the state of the parser.

S280b. ⟨*streams that issue two forms of prompts* S279a⟩+≡ (S260) ◁S280a

```
interactiveParsedStream : 't lexer * ('t, 'a) polyparser -> string * line stream * pr
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
```

J.5 Further reading

Fat book by Aho and Ullman (1972).

Really nice paper by Knuth (1965).

Wirth (1977) master of the hand-written recursive-descent parser.

Gibbons and Jones (1998)

Ramsey (1999)

Mcbride and Paterson (2008)

J Lexical analysis, parsing, and reading using ML

VIII. The Supporting Cast

Chapter 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 three pieces left over—code that is used in Impcore, is not shared in any other interpreter, and is not parsing:

- Code that runs unit tests
- Printing functions
- The implementation of function environments.

There are so few pieces that they don't warrant a lot of organization and description. But they are not all equally worth reading:

- The unit-testing piece is interesting; this is the source of truth about what it means to pass a unit test and how unit tests are run. (A version for μScheme, which is very similar to this one, appears in Section L.6.) 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.
- The printing functions may be of minor interest, if for example you want to write your own. But once you've seen a couple, you've seen them all.
- 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

Creating abstract syntax

To make these structures easy to create, I define a creator function for each alternative in the sum, as well as for `Userfun`.

S287. ⟨*function prototypes for Impcore* S287⟩≡ (S290) S289a ▷

```
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);
```

Extended definitions

But as discussed in the sidebar on page 25, Impcore also has *extended definitions*, which include unit tests. If you like, you can just use extended definitions and not worry about how they are implemented. But if you want to understand their implementations, you'll need to start with these descriptions of how extended definitions and unit tests are represented:

S288a. ⟨*xdef.t* S288a⟩≡
```
XDef*     = DEF          (Def)
          | USE          (Name)
          | TEST         (UnitTest)
UnitTest* = CHECK_EXPECT (Exp check, Exp expect)
          | CHECK_ASSERT (Exp)
          | CHECK_ERROR  (Exp)
```
To remember all the unit tests in a file, I use a list.

S288b. ⟨*type definitions for Impcore* S288b⟩≡ (S290) S288c ▷
```
typedef struct UnitTestlist *UnitTestlist; // list of UnitTest
```
A `UnitTestlist` is list of pointers of type `UnitTest`. I use this naming convention in all my C code. List types are manifest, and their definitions are in the lists interface in chunk 46a. I also define a type for lists of `Exp`s.

S288c. ⟨*type definitions for Impcore* S288b⟩+≡ (S290) ◁ S288b
```
typedef struct Explist *Explist; // list of Exp
```

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 we have a source of extended definitions. The details are relegated to Appendix F.

A source of extended definitions is called an `XDefstream`. To obtain the next definition from such a source, call `getxdef`. Function `getxdef` returns either a pointer to the next definition or, if the source is exhausted, the `NULL` pointer. And if there is some problem converting input to abstract syntax, `getxdef` may call `synerror` (page S289).

S288d. ⟨*shared type definitions* S288d⟩≡ (S290) S288g ▷
```
typedef struct XDefstream *XDefstream;
```
S288e. ⟨*shared function prototypes* S288e⟩≡ (S290) S288f ▷
```
XDef getxdef(XDefstream xdefs);
```
To create a stream of definitions, we need a source of lines. That source can be a string compiled into the program, or an external file. So that error messages can refer to the source, we need to give its name. And if the source is a file, we need to say whether to prompt for input. (Reading from an internal string never prompts.)

S288f. ⟨*shared function prototypes* S288e⟩+≡ (S290) ◁ S288e S289c ▷
```
XDefstream stringxdefs(const char *stringname, const char *input);
XDefstream filexdefs  (const char *filename, FILE *input, Prompts prompts);
```
Prompts are either absent or standard; the interface provides no way to change prompts.

S288g. ⟨*shared type definitions* S288d⟩+≡ (S290) ◁ S288d S289b ▷
```
typedef enum Prompts { NO_PROMPTS, STD_PROMPTS } 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 remembered unit tests.

S289a. ⟨*function prototypes for Impcore* S287⟩+≡ (S290) ◁ S287 S291b ▷

```
void readevalprint(XDefstream s, Valenv globals, Funenv functions, Echo echo_level);
```

As with `evaldef`, the `echo_level` parameter controls whether `readevalprint` prints the values and names of top-level expressions and functions.

S289b. ⟨*shared type definitions* S288d⟩+≡ (S290) ◁ S288g S289d ▷

```
typedef enum Echo { NO_ECHOES, ECHOES } Echo;
```

Interface to the extensible printer

The implementations of `print` and `fprint` are *extensible*; adding a new conversion specification is as simple as calling `installprinter`:

S289c. ⟨*shared function prototypes* S288e⟩+≡ (S290) ◁ S288f S289f ▷

```
void installprinter(unsigned char c, Printer *take_and_print);
```

The conversion specifications listed above are installed when the interpreter launches, by code chunk ⟨*install conversion specifications for* `print` *and* `fprint` S297e⟩. The details, including the definition of `Printer`, are in Sections F.3 and K.3.

Complexities of error signaling

The `Sourceloc` values are taken care of by the parsing infrastructure described in Appendix G, which is the place from which `synerror` is called.

S289d. ⟨*shared type definitions* S288d⟩+≡ (S290) ◁ S289b S289e ▷

```
typedef struct Sourceloc *Sourceloc;
```

The possibility of printing source-code locations complicates the interface. When the 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, it's a different story—it's useful to have the file's name and the number of the line containing the bad syntax. But the error module doesn't know where the interpreter is reading code from—only the `main` function in chunk S292a knows that. So the error module has to be told how syntax errors should be formatted: with locations or without.

S289e. ⟨*shared type definitions* S288d⟩+≡ (S290) ◁ S289d S295b ▷

```
typedef enum ErrorFormat { WITH_LOCATIONS, WITHOUT_LOCATIONS } ErrorFormat;
```

S289f. ⟨*shared function prototypes* S288e⟩+≡ (S290) ◁ S289c S294e ▷

```
void set_toplevel_error_format(ErrorFormat format);
```

`filexdefs`	S186b
`type Funenv`	44f
`getxdef`	S186c
`type Printer`	S189b
`readevalprint`	S291a
`set_toplevel_error_format`	S195a
`stringxdefs`	S186b
`type Valenv`	44f
`type XDef`	$\mathcal{A}$

K.1.1 Interfaces and the master 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 six 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, we need not worry about getting things in the right order.

To make it possible to reuse the general-purpose interfaces in later interpreters, I also distinguish between shared and unshared definitions; a definition is "shared" if it is used in another interpreter later in the book.

S290. ⟨`all.h` *for Impcore* S290⟩≡

```
#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


⟨type definitions for Impcore S288b⟩
⟨shared type definitions S288d⟩

⟨structure definitions for Impcore S204b⟩
⟨shared structure definitions S178d⟩

⟨function prototypes for Impcore S287⟩
⟨shared function prototypes S288e⟩

⟨macro definitions used in parsing S205c⟩
⟨declarations of global variables used in lexical analysis and parsing S211h⟩
```

K.1.2 Additional implementations

K.1.3 Evaluation of extended definitions

As shown on page S288, the `XDef` type includes both ordinary and extended definitions, and an `XDefstream` provides a stream of `XDef`s, usually from a file or from a user's input.

Responsibility for evaluating definitions is shared 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`.

The true definitions are passed on to `evaldef`.

S291a. ⟨*eval.c* S291a⟩≡

```
void readevalprint(XDefstream xdefs, Valenv globals, Funenv functions, Echo echo) {
    UnitTestlist pending_unit_tests = NULL;  // to be 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 S291c⟩
            break;
        case DEF:
            evaldef(d->def, globals, functions, echo);
            break;
        default:
            assert(0);
        }

    process_tests(pending_unit_tests, globals, functions);
}
```

Function `process_tests`, defined in Section K.2 on page S294, runs the `pending_unit_te` in the order in which they appear in the source code.

S291b. ⟨*function prototypes for Impcore* S287⟩+≡ (S290) ◁S289a S294d▷

```
void process_tests(UnitTestlist tests, Valenv globals, Funenv functions);
```

On seeing `use`, we open the file named by `use`, build a stream of definitions, and through `readevalprint`, recursively call `evaldef` on all the definitions in that file. When reading definitions via `use`, the interpreter neither prompts nor echoes.

S291c. ⟨*evaluate* `d->use`*, possibly mutating* `globals` *and* `functions` S291c⟩≡ (S291a)

```
{
    const char *filename = nametostr(d->use);
    FILE *fin = fopen(filename, "r");
    if (fin == NULL)
        runerror("cannot open file \"%s\"", filename);
    readevalprint(filexdefs(filename, fin, NO_PROMPTS), globals, functions, echo
    fclose(fin);
}
```


As noted in Exercise 35, this code can leak open file descriptors.

K.1.4 *Implementation of* `main`

The `main` function coordinates all the pieces and forms a working interpreter. Such an interpreter can operate in two modes:

- In *interactive* mode, the interpreter prompts for every input, and when it detects a syntax error, it does not print the source-code location.

- In *non-interactive* mode, the interpreter does not prompt for any input, and when it detects a syntax error, it prints the source-code locations.

Interactive mode is meant for interactive use, and non-interactive mode is meant for redirecting standard input from a file. The interpreter is in interactive mode by default, but if its given the option -q, for "quiet," it operates in non-interactive mode.

S292a. ⟨*impcore.c* S292a⟩≡
```
int main(int argc, char *argv[]) {
    bool interactive  = (argc <= 1) || (strcmp(argv[1], "-q") != 0);
    Prompts prompts  = interactive ? STD_PROMPTS : NO_PROMPTS;
    set_toplevel_error_format(interactive ? WITHOUT_LOCATIONS : WITH_LOCATIONS);

    ⟨install conversion specifications for print and fprint S297e⟩

    Valenv globals   = mkValenv(NULL, NULL);
    Funenv functions = mkFunenv(NULL, NULL);
    ⟨install the initial basis in functions S293a⟩

    XDefstream xdefs = filexdefs("standard input", stdin, prompts);

    while (setjmp(errorjmp))
        ;
    readevalprint(xdefs, globals, functions, ECHOES);
    return 0;
}
```

Before entering its main loop, the interpreter performs these phases of initialization:

- It decides whether it is operating interactively or non-interactively, and it sets prompts and the error format accordingly.
- It initializes `print` and `fprint` (the code appears in Appendix K).
- It creates empty environments for functions and global variables, then populates the `functions` environment with functions from the initial basis.
- It creates a stream of `XDefs` from the standard input.

The main loop is in the `readevalprint` function, the call to which is preceded by a C idiom:

S292b. ⟨*idiomatic error handler* S292b⟩≡
```
while (setjmp(errorjmp)) {
    ⟨recover from an error⟩
}
```

This idiom uses `setjmp` to deal with errors. On the first loop test, `setjmp` initializes `errorjmp` and returns zero, so the code in ⟨*recover from an error*⟩ is not executed, and control continues following the `while` 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. (In the definition of `main` above, no work is needed to recover from an error, instead of a block containing the action ⟨*recover from an error*⟩, I use an empty statement, which is written as a single semicolon.) On the next iteration through the `while` statement, the process starts over from the beginning, because `setjmp` resets the jump buffer and returns zero again.

The initial basis includes both primitives and user-defined functions. We install the primitives first.

S293a. ⟨*install the initial basis in* `functions` S293a⟩≡ (S292a) S293c ▷

```
{
    static const char *prims[] =
      { "+", "-", "*", "/", "<", ">", "=", "println", "print", "printu", 0 };
    for (const char **p = prims; *p; p++) {
        Name x = strtoname(*p);
        bindfun(x, mkPrimitive(x), functions);
    }
}
```

I represent the user-defined part of the initial basis as a single string, which is interpreted by `readevalprint`. These functions also appear in Figure 1.3 on page 27, from which this code is derived automatically.

S293b. ⟨*predefined Impcore functions, as strings* S293b⟩≡ (S293c)

```
        "(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"
```

S293c. ⟨*install the initial basis in* `functions` S293a⟩+≡ (S292a) ◁ S293a

```
{
    const char *fundefs =
      ⟨predefined Impcore functions, as strings S293b⟩;
    if (setjmp(errorjmp))
        assert(0); // if error in predefined function, die horribly
    readevalprint(stringxdefs("predefined functions", fundefs), globals, functic
}
```

K.1.5 Implementation of names

Because names and environments are core concepts in programming languages, their implementations are included in this chapter. The implementations are straightforward, and the techniques I use should be familiar.

Each name is associated with a string. I just store the string inside the name.

S293d. ⟨*name.c* S293d⟩≡ S293e ▷

```
struct Name {
    const char *s;
};
```

Returning the string associated with a name is trivial.

S293e. ⟨*name.c* S293d⟩+≡ ◁ S293d S294a ▷

```
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 I get a string I have seen before, I must return the same name I returned before.

To remember what I have seen and returned, I use the simplest possible data structure: `all_names`, a list of all names we ever returned. Given a string `s`, a simple linear search finds the name associated with it, if any.

S294a. ⟨*name.c* S293d⟩+≡ ◁ S293e

```
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 S294b⟩
}
```

A faster implementation might use a search tree or a hash table, not a simple list. Hanson (1996, Chapter 3) shows such an implementation.

If the string `s` isn't associated with any name on the list `all_names`, I make a new name and add it.

S294b. ⟨*allocate a new name, add it to* `all_names`*, and return it* S294b⟩≡ (S294a)

```
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;
```

K.2 Running unit tests

Running a list of unit tests is the job of the function `process_tests`:

S294c. ⟨*imptests.c* S294c⟩≡ S295a ▷

```
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.

S294d. ⟨*function prototypes for Impcore* S287⟩+≡ (S290) ◁ S291b S295c ▷

```
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.5 on page S196.

S294e. ⟨*shared function prototypes* S288e⟩+≡ (S290) ◁ S289f S297f ▷

```
void report_test_results(int npassed, int ntests);
```

The key fact about the testing interface is that the list of tests coming in contains the last test first, but we must run the first test 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.

S295a. ⟨*imptests.c* S294c⟩+≡ ◁S294c S295d▷

```
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 by hand, so we probably don't have to worry about more than dozens of tests, for which default stack space should be adequate.

The heavy lifting is done by function `test_result`, which returns a value of type `TestResult`.

S295b. ⟨*shared type definitions* S288d⟩+≡ (S290) ◁S289e

```
typedef enum TestResult { TEST_PASSED, TEST_FAILED } TestResult;
```

S295c. ⟨*function prototypes for Impcore* S287⟩+≡ (S290) ◁S294d

```
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.

S295d. ⟨*imptests.c* S294c⟩+≡ ◁S295a

```
TestResult test_result(UnitTest t, Valenv globals, Funenv functions) {
    switch (t->alt) {
    case CHECK_EXPECT:
        ⟨run check-expect test t, returning TestResult S295e⟩
    case CHECK_ASSERT:
        ⟨run check-assert test t, returning TestResult S296a⟩
    case CHECK_ERROR:
        ⟨run check-error test t, returning TestResult S296b⟩
    default:
        assert(0);
    }
}
```

To run a `check-expect`, we evaluate both the "check" and "expect" expressions, each under the protection of an error handler. If an error occurs under either evaluation, the test fails. Otherwise we compare the values `check` and `expect`. If they differ, the test fails; if not, the test passes. All failures trigger error messages.

S295e. ⟨*run* `check-expect` *test* `t`*, returning* `TestResult` S295e⟩≡ (S295d)

```
{   Valenv empty_env = mkValenv(NULL, NULL);
    if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_expect.check failed with an error S296d⟩
        bufreset(errorbuf);
        return TEST_FAILED;
    }
    Value check = eval(t->check_expect.check, globals, functions, empty_env);
```

`_setjmp`	$\mathcal{B}$
`bufreset`	S186f
`errorbuf`	S193a
`eval`	45e
`type Funenv`	44f
`lengthUL`	$\mathcal{A}$
`mkNL`	$\mathcal{A}$
`type Name`	43b
`type Namelist`	43b
`report_test_results`	S196c
`set_error_mode`	S193a
`testjmp`	S193a
`type UnitTest`	$\mathcal{A}$
`type UnitTestlist`	S288b
`type Valenv`	44f
`type Value`	44a

```
        if (setjmp(testjmp)) {
            ⟨report that evaluating t->check_expect.expect failed with an error S297a⟩
            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 S296c⟩
            return TEST_FAILED;
        } else {
            return TEST_PASSED;
        }
    }
```

To run a check-assert, we evaluate just one expression, which should evaluate, without error, to a nonzero value.

S296a. ⟨*run* check-assert *test* t*, returning* TestResult S296a⟩≡ (S295d)

```
    {   Valenv empty_env = mkValenv(NULL, NULL);
        if (setjmp(testjmp)) {
            ⟨report that evaluating t->check_assert failed with an error S297c⟩
            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 S297b⟩
            return TEST_FAILED;
        } else {
            return TEST_PASSED;
        }
    }
```

To run a check-error, we use the same tools in different ways. Again we evaluate an expression under the protection of an error handler, but now, if an error occurs, the test passes. If not, it fails.

S296b. ⟨*run* check-error *test* t*, returning* TestResult S296b⟩≡ (S295d)

```
    {   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 S297d⟩
        return TEST_FAILED;
    }
```

Error-reporting code is voluminous but uninteresting.

S296c. ⟨*report failure because the values are not equal* S296c⟩≡ (S295e)

```
    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);
```

S296d. ⟨*report that evaluating* t->check_expect.check *failed with an error* S296d⟩≡ (S295e)

```
    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));
```

S297a. ⟨*report that evaluating* `t->check_expect.expect` *failed with an error* S297a⟩≡ (S295e)

```
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));
```

S297b. ⟨*report failure because the value is zero* S297b⟩≡ (S296a)

```
fprint(stderr, "Check-assert failed: %e evaluated to 0.\n", t->check_assert);
```

S297c. ⟨*report that evaluating* `t->check_assert` *failed with an error* S297c⟩≡ (S296a)

```
fprint(stderr, "Check-assert failed: evaluating %e causes an error: %s.\n",
               t->check_assert, bufcopy(errorbuf));
```

S297d. ⟨*report that evaluating* `t->check_error` *produced* `check` S297d⟩≡ (S296b)

```
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 Printing functions

Table 1.6 on 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. That work is done here:

S297e. ⟨*install conversion specifications for* `print` *and* `fprint` S297e⟩≡ (S292a)

```
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.3.3 on page S191. Functions that print lists are generated automatically. The remaining functions, which print Impcore's abstract syntax and values, are defined here.

S297f. ⟨*shared function prototypes* S288e⟩+≡ (S290) ◁S294e

```
Printer printexp, printdef, printvalue, printfun;
```

Function `printexp` reverses the process of parsing: it renders abstract syntax into concrete syntax.

S297g. ⟨*printfuns.c* S297g⟩≡ S298a ▷

```
void printexp(Printbuf output, va_list_box *box) {
    Exp e = va_arg(box->ap, Exp);
    if (e == NULL) {
        bprint(output, "<null>");
        return;
```

`_setjmp`	$\mathcal{B}$
`bprint`	S188f
`bufreset`	S186f
`errorbuf`	S193a
`eval`	45e
type `Exp`	$\mathcal{A}$
`functions`	S295d
`globals`	S295d
`installprinter`	S189a
type `Printbuf`	S186d
`printchar`	S191c
`printdecimal`	S191c
`printdef`	S298a
type `Printer`	S189b
`printexplist`	$\mathcal{A}$
`printfun`	S299c
`printname`	S191c
`printnamelist`	$\mathcal{A}$
`printpar`	S192d
`printparlist`	$\mathcal{A}$
`printpercent`	S191c
`printstring`	S191c
`printvalue`	S299b
`printvaluelist`	$\mathcal{A}$
`testjmp`	S193a
type `va_list_box`	S189c
type `Valenv`	44f
type `Value`	44a

```
    }

    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.

S298a. ⟨*printfuns.c* S297g⟩+≡ ◁ S297g S298b ▷

```
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, here is a function that prints extended definitions.

S298b. ⟨*printfuns.c* S297g⟩+≡ ◁ S298a S299b ▷

```
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 S299a⟩
        break;
    case DEF:
        bprint(output, "%t", d->def);
        break;
    }
    assert(0);
}
```

S299a. ⟨*print unit test* d->test *to file* output S299a⟩≡ (S298b)

```
{   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.

S299b. ⟨*printfuns.c* S297g⟩+≡ ◁S298b S299c▷

```
void printvalue(Printbuf output, va_list_box *box) {
    Value v = va_arg(box->ap, Value);
    bprint(output, "%d", v);
}
```

bprint	S188f
type Def	$\mathcal{A}$
type Func	$\mathcal{A}$
type Printbuf	S186d
type UnitTest	$\mathcal{A}$
type va_list_box	S189c
type Value	44a
type XDef	$\mathcal{A}$

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.

S299c. ⟨*printfuns.c* S297g⟩+≡ ◁S299b

```
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);
    }
```

```
}
```

K.4 Printing primitives

S300a. ⟨*apply Impcore primitive* `println` *to* `vs` *and return* S300a⟩≡ (52c)

```
{
    checkargc(e, 1, lengthVL(vs));
    Value v = nthVL(vs, 0);
    print("%v\n", v);
    return v;
}
```


S300b. ⟨*apply Impcore primitive* `printu` *to* `vs` *and return* S300b⟩≡ (52c)

```
{
    checkargc(e, 1, lengthVL(vs));
    Value v = nthVL(vs, 0);
    print_utf8(v);
    return v;
}
```

K.5 Implementation of function environments

This code is continued from Chapter 1, which gives the implementation of value environments. Except for types, the code is identical to code in Section 1.6.3 on page 55.

S300c. ⟨*env.c* S300c⟩≡ S300d ▷

```
struct Funenv {
    Namelist xs;
    Funclist funs;
    // invariant: both lists are the same length
};
```

S300d. ⟨*env.c* S300c⟩+≡ ◁ S300c S300e ▷

```
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;
}
```

S300e. ⟨*env.c* S300c⟩+≡ ◁ S300d S300f ▷

```
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;
}
```

S300f. ⟨*env.c* S300c⟩+≡ ◁ S300e S301a ▷

```
bool isfunbound(Name name, Funenv env) {
    return findfun(name, env) != NULL;
}
```

S301a. ⟨*env.c* S300c⟩+≡ ◁ S300f S301b ▷

```
Func fetchfun(Name name, Funenv env) {
    Func *fp = findfun(name, env);
    assert(fp != NULL);
    return *fp;
}
```

S301b. ⟨*env.c* S300c⟩+≡ ◁ S301a S301c ▷

```
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);
    }
}
```

S301c. ⟨*env.c* S300c⟩+≡ ◁ S301b

```
void dump_fenv_names(Funenv env) {
    Namelist xs;
    if (env)
        for (xs = env->xs; xs; xs = xs->tl)
            print("%n\n", xs->hd);
}
```

Chapter contents

Supporting code for μScheme

L

The stars of the μScheme show are presented in Chapter 2. Here you'll find the supporting cast. In addition to code for implementing environments, for parsing μScheme, and for running unit tests, all of which is similar to the 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+ in Chapter 3.

L.1 Excerpts from the interpreter

S303a. ⟨*ast.t* S303a⟩≡
```
XDef* = DEF    (Def)
      | USE    (Name)
      | TEST   (UnitTest)

UnitTest* = CHECK_EXPECT (Exp check, Exp expect)
          | CHECK_ASSERT (Exp)
          | CHECK_ERROR  (Exp)
```

S303b. ⟨*type definitions for μScheme* S303b⟩≡ (S303d) S306d ▷
```
typedef struct UnitTestlist  *UnitTestlist;  // list of UnitTest
typedef struct Explist  *Explist;            // list of Exp
```

S303c. ⟨*early type definitions for μScheme* S303c⟩≡ (S303d)
```
typedef struct Valuelist *Valuelist;     // list of Value
```

MISSING: RELEGATED DEFINITIONS OF PREDEFINED LIST FUNCTIONS (`caaar`, `list5`, and friends).

As in Impcore, I gather all the interfaces into a single C header file.

S303d. ⟨`all.h` *for μScheme* S303d⟩≡
```
#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 S303c⟩
⟨type definitions for μScheme S303b⟩
⟨shared type definitions 43b⟩

⟨structure definitions for μScheme S313b⟩
⟨shared structure definitions S206a⟩

⟨function prototypes for μScheme S304a⟩
⟨shared function prototypes S306c⟩

⟨macro definitions used in parsing S205c⟩
⟨declarations of global variables used in lexical analysis and parsing S211h⟩
```

Allocation

Before the first call to `allocate`, a client must call `initallocate`. For reasons that aren't discussed until Chapter 4, `initallocate` is given a pointer to the environment containing the global variables.

S304a. ⟨*function prototypes for μScheme* S304a⟩≡ (S303d) S304b ▷

```
void initallocate(Env *globals);
```

Values

Before executing any code that refers to `truev` or `falsev`, clients must call `initvalue`.

S304b. ⟨*function prototypes for μScheme* S304a⟩+≡ (S303d) ◁ S304a S304c ▷

```
void initvalue(void);
```

Read-eval-print loop

To handle a sequence of extended definitions, we use `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; instead it calls `synerror` or `runerror`. And if an error occurs, we don't want to lose the definitions that precede it. So instead of returning a new environment, `readevalprint` writes the new environment through an environment *pointer* `envp`, which is passed as a parameter.

S304c. ⟨*function prototypes for μScheme* S304a⟩+≡ (S303d) ◁ S304b S304d ▷

```
void readevalprint(XDefstream xdefs, Env *envp, Echo echo);
```

Primitives

Compared to Impcore, μScheme has many primitives. The function `addprimitives` mutates an existing environment by adding bindings to all the primitive operations.

S304d. ⟨*function prototypes for μScheme* S304a⟩+≡ (S303d) ◁ S304c S305a ▷

```
void addprimitives(Env *envp);
```

Here are some of the printing functions used to implement `print` and `fprint`.

S305a. ⟨*function prototypes for μScheme* S304a⟩+≡ (S303d) ◁S304d S306b▷

```
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.1.1 Implementation of the evaluator

S305b. ⟨*eval.c declarations* S305b⟩≡

```
static Valuelist evallist(Explist es, Env env);
```

S305c. ⟨*if* `echo` *calls for printing, print either* `v` *or the bound name* S305c⟩≡ (162a)

```
if (echo == ECHOES) {
    if (d->val.exp->alt == LAMBDAX)
        print("%n\n", d->val.name);
    else
        print("%v\n", v);
}
```

S305d. ⟨*if* `echo` *calls for printing, print* `v` S305d⟩≡ (162b)

```
if (echo == ECHOES)
    print("%v\n", v);
```

Function `readevalprint` evaluates definitions, updates the environment `*envp`, and remembers unit tests. After all definitions have been read, it runs the remembered unit tests. The last test added to `unit_tests` is the one at the front of the list, but we want to run tests in the order in which they appear, so the tests are run back to front.

S305e. ⟨*evaldef.c* S305e⟩≡

```
void readevalprint(XDefstream xdefs, Env *envp, Echo echo) {
    UnitTestlist pending_unit_tests = NULL;

    for (XDef d = getxdef(xdefs); d; d = getxdef(xdefs)) {
        ⟨lower definition d as needed S305f⟩
        switch (d->alt) {
        case DEF:
            *envp = evaldef(d->def, *envp, echo);
            break;
        case USE:
            ⟨read in a file and update *envp S306a⟩
            break;
        case TEST:
            pending_unit_tests = mkUL(d->test, pending_unit_tests);
            break;
        default:
            assert(0);
        }
    }

    process_tests(pending_unit_tests, *envp);
}
```

S305f. ⟨*lower definition* `d` *as needed* S305f⟩≡ (S305e)

```
/* not in uScheme */
```

In the DEF case, as alluded to on page S304, the assignment to *envp 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. This code is more complicated than the analogous code in Impcore: Impcore's readevalprint simply mutates the global environment. In μScheme, environments are not mutable, so we mutate a C location instead.

Reading a file is as in Impcore, except that again we cannot mutate an environment, so we mutate *envp instead. When readevalprint calls itself recursively to read a file, it passes the same envp it was given.

S306a. ⟨*read in a file and update* *envp S306a⟩≡ (S305e)

```
{
    const char *filename = nametostr(d->use);
    FILE *fin = fopen(filename, "r");
    if (fin == NULL)
        runerror("cannot open file \"%s\"", filename);
    readevalprint(filexdefs(filename, fin, NO_PROMPTS), envp, echo);
    fclose(fin);
}
```

Unit tests are run by code in Section L.6.

S306b. ⟨*function prototypes for μScheme* S304a⟩+≡ (S303d) ◁S305a S307d▷

```
void process_tests(UnitTestlist tests, Env rho);
```

L.1.2 Primitives

S306c. ⟨*shared function prototypes* S306c⟩≡ (S303d) S313d▷

```
Primitive arith, binary, unary;
```

To define the primitives and associate each one with its tag and function, I resort to macro madness. 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 this technique ensures there is a single point of truth about the primitives (that point of truth is the file prim.h), which helps guarantee that the enumeration type is consistent with the initialization code.

S306d. ⟨*type definitions for μScheme* S303b⟩+≡ (S303d) ◁S303b

```
enum {
  #define xx(NAME, TAG, FUNCTION) TAG,
  #include "prim.h"
  #undef xx
  UNUSED_TAG
};
```

In addprimitives, the xx macro extends the initial environment.

S306e. ⟨*install primitive functions into* env S306e⟩≡ (S309a)

```
#define xx(NAME, TAG, FUNCTION) \
    env = bindalloc(strtoname(NAME), mkPrimitive(TAG, FUNCTION), env);
#include "prim.h"
#undef xx
```

S306f. ⟨*JUNK prim.c* S306f⟩≡

```
Env primenv(void) {
    Env env = NULL;
    #define xx(NAME, TAG, FUNCTION) \
        env = bindalloc(strtoname(NAME), mkPrimitive(TAG, FUNCTION), env);
    #include "prim.h"
```

```
    #undef xx
    return env;
}
```

Arithmetic primitives

These are the arithmetic primitives.

S307a. ⟨*prim.h* S307a⟩≡ S307c ▷

```
xx("+", PLUS,  arith)
xx("-", MINUS, arith)
xx("*", TIMES, arith)
xx("/", DIV,   arith)
xx("<", LT,    arith)
xx(">", GT,    arith)
```

We need special support for division, because while μScheme requires that division round toward minus infinity, C guarantees only that dividing positive operands rounds toward zero.

S307b. ⟨*prim.c* S307b⟩≡ S307e ▷

```
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

S307c. ⟨*prim.h* S307a⟩+≡ ◁ S307a S308b ▷

```
xx("cons", CONS, binary)
xx("=",    EQ,   binary)
```

I implement them with the function `binary`, which delegates to `cons` and `equalatoms`.

S307d. ⟨*function prototypes for µScheme* S304a⟩+≡ (S303d) ◁ S306b S316b ▷

```
Value cons(Value v, Value w);
Value equalatoms(Value v, Value w);
```

S307e. ⟨*prim.c* S307b⟩+≡ ◁ S307b S308a ▷

```
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);
```

```
    }
}
```

The implementation of 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 call the C function `equalatoms`. A different function, `equalpairs`, is used in Section L.6 to implement `check-expect`.

S308a. ⟨*prim.c* S307b⟩+≡ ◁S307e

```
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

S308b. ⟨*prim.h* S307a⟩+≡ ◁S307c

```
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)
```

S308c. ⟨*other cases for unary primitives* S308c⟩≡ (164a)

```
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);
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.1.3 *Implementation of the interpreter's* `main` *procedure*

As in the Impcore interpreter, `main` processes arguments, initializes the interpreter, and runs the read-eval-print loop.

S309a. ⟨*scheme.c* S309a⟩≡

```
int main(int argc, char *argv[]) {
    bool interactive = (argc <= 1) || (strcmp(argv[1], "-q") != 0);
    Prompts prompts = interactive ? STD_PROMPTS : NO_PROMPTS;
    set_toplevel_error_format(interactive ? WITHOUT_LOCATIONS : WITH_LOCATIONS);

    initvalue();

    ⟨install printers S309b⟩

    Env env = NULL;
    initallocate(&env);
    ⟨install primitive functions into env S306e⟩
    ⟨install predefined functions into env S310a⟩

    XDefstream xdefs = filexdefs("standard input", stdin, prompts);

    while (setjmp(errorjmp))
        ;
    readevalprint(xdefs, &env, ECHOES);
    return 0;
}
```

We have many printers.

S309b. ⟨*install printers* S309b⟩≡ (S309a)

```
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);
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* S310e⟩.

S310a. ⟨*install predefined functions into* `env` S310a⟩≡ (S309a)
```
const char *fundefs = ⟨predefined μScheme functions, as strings (from ⟨predefined μScheme functions 98a⟩)⟩;
if (setjmp(errorjmp))
    assert(0);  // fail if error occurs in predefined functions
readevalprint(stringxdefs("predefined functions", fundefs), &env, NO_ECHOES);
```

L.1.4 Memory allocation

To use `malloc` requires no special initialization or resetting.

S310b. ⟨*loc.c* S310b⟩≡
```
void initallocate(Env *globals) {
    (void)globals;
}
```

L.2 μSCHEME CODE NOT INCLUDED IN CHAPTER 2

Function `sqrt` produces the largest integer that is not greater than the square root of `n`. This is a pathetic definition of square root, but it does work on perfect squares, and it's also useful for testing primality.

S310c. ⟨*definition of* `sqrt` S310c⟩≡ (120a)
```
-> (define sqrt (n)
     (letrec ((find (lambda (r)
                      (if (> (* r r) n) (- r 1) (find (+ r 1))))))
       (find 0)))
```

Next is a scurvy Noweb trick; by extending the definition of ⟨*transcript* S310d⟩ in this appendix, I expose ⟨*polymorphic-set transcript* 135b⟩ to my testing software, while preventing the definitions in ⟨*polymorphic-set transcript* 135b⟩ from interfering with non-polymorphic uses of the set operations.

S310d. ⟨*transcript* S310d⟩≡
⟨*polymorphic-set transcript* 135b⟩

Unicode code points

S310e. ⟨*predefined μScheme functions* S310e⟩≡ S310f ▷
```
(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

We add additional integer operations, all of which are defined exactly as they would be in Impcore. We begin with comparisons.

S310f. ⟨*predefined μScheme functions* S310e⟩+≡ ◁ S310e S311a ▷
```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

We continue with min and max.

S311a. ⟨*predefined μScheme functions* S310e⟩+≡ ◁S310f S311b▷
```
(define max (x y) (if (> x y) x y))
(define min (x y) (if (< x y) x y))
```

Finally, we add negation, modulus, greatest common divisor, and least common multiple.

S311b. ⟨*predefined μScheme functions* S310e⟩+≡ ◁S311a S311f▷
```
(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

S311c. ⟨*more predefined combinations of* car *and* cdr S311c⟩≡ S311d▷
```
(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)))
```

S311d. ⟨*more predefined combinations of* car *and* cdr S311c⟩+≡ ◁S311c S311e▷
```
(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)))
```

S311e. ⟨*more predefined combinations of* car *and* cdr S311c⟩+≡ ◁S311d
```
(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)))
```

_setjmp	$\mathcal{B}$
type Env	155a
env	S309a
errorjmp	47
readevalprint	S304c
stringxdefs	S288f

S311f. ⟨*predefined μScheme functions* S310e⟩+≡ ◁S311b
```
(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 Implementation of μScheme environments

μScheme environments are significantly different from Impcore environments, but not so dramatically different that it's worth putting a very similar 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 used in Impcore environments, and it militates toward a different representation.

First, and most important, environments are immutable, as we can see from the interface in Section 2.12.2 on page 155. The operational semantics never mutates an environment, and there is really no need, because all the mutation is done on locations. Moreover, if we wanted to mutate environments, it wouldn't be safe to copy them just by copying pointers; this 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.

S312a. ⟨*env.c* S312a⟩≡ S312b ▷

```
struct Env {
    Name name;
    Value *loc;
    Env tl;
};
```

We look up a name by following `tl` pointers.

S312b. ⟨*env.c* S312a⟩+≡ ◁ S312a S312c ▷

```
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. There is never any mutation.

S312c. ⟨*env.c* S312a⟩+≡ ◁ S312b S312d ▷

```
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.

S312d. ⟨*env.c* S312a⟩+≡ ◁ S312c S312e ▷

```
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;
}
```

In case it helps you debug your code, you might want to print environments. Here is a printing function `printenv`.

S312e. ⟨*env.c* S312a⟩+≡ ◁ S312d S313a ▷

```
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, " }");
}
```

To help support static analysis of μScheme programs, we can dump all the names in an environment.

S313a. ⟨*env.c* S312a⟩+≡ ◁S312e

```
void dump_env_names(Env env) {
    for ( ; env; env = env->tl)
        fprint(stdout, "%n\n", env->name);
}
```


L.4 Parsing μScheme code

L.4.1 Parsing tables and reduce functions

Here are all the components that go into μScheme's abstract syntax. They include all the components used to parse Impcore, plus a `Value` component that is used when parsing a quoted S-expression.

S313b. ⟨*structure definitions for μScheme* S313b⟩≡ (S303d)

```
struct Component {
    Exp exp;
    Explist exps;
    Name name;
    Namelist names;
    Value value;
    ⟨fields of μScheme Component added in exercises S315c⟩
};
```

Here is the usage table for the parenthesized keywords.

S313c. ⟨*parse.c* S313c⟩≡ S314a ▷

```
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 S315h⟩
    { -1, NULL }
};
```

Shift functions are as in Impcore, but with two additions: to parse quoted S-expressions, shift function `sSexp` has been added, and to parse bindings in `LETX` forms, `sBindings` has been added.

S313d. ⟨*shared function prototypes* S306c⟩+≡ (S303d) ◁S306c S315i ▷

allocate	156a
bindalloc	155c
bprint	S188f
type Env	155a
type Exp	$\mathcal{A}$
type Explist	S303b
type Name	43b
type Namelist	43b
type ParserResult	S207c
type ParserState	S206b
type Printbuf	S186d
sBindings	S316a
sSexp	S315k
type va_list_box	S189c
type Value	$\mathcal{A}$
type Valuelist	S303c

```
ParserResult sSexp    (ParserState state);
ParserResult sBindings(ParserState state);
```

Using the new shift functions, here is the `exptable`, for parsing expressions.

S314a. ⟨*parse.c* S313c⟩+≡ ◁S313c S314c▷

```
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 S315d⟩
⟨lowering functions for μScheme+ S329d⟩

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+ generated automatically⟩
  ⟨rows added to μScheme's exptable in exercises S315e⟩
  { NULL,     ANEXP(APPLY),   applyshifts }  // must come last
};
```

S314b. ⟨*rows of μScheme's* `exptable` *that are sugared in μScheme+* **[[uscheme]]** S314b⟩≡

```
  { "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 Section F.1.2 on page S182.

S314c. ⟨*parse.c* S313c⟩+≡ ◁S314a S314d▷

```
bool read_tick_as_quote = true;
```

The codes used in `exptable` tell `reduce_to_exp` how to reduce components to an expression.

S314d. ⟨*parse.c* S313c⟩+≡ ◁S314c S315a▷

```
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+ generated automatically⟩
    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 S315f⟩
    }
    assert(0);
}
```

S314e. ⟨*cases for* `reduce_to_exp` *that are sugared in μScheme+* **[[uscheme]]** S314e⟩≡

```
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.

S315a. ⟨*parse.c* S313c⟩+≡ ◁S314d S315k▷

```
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) generated automatically⟩
    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 S315g⟩
    default:           assert(0);  // incorrectly configured parser
    }
}
```


S315b. ⟨`reduce_to_xdef` *case for* `ADEF(DEFINE)` **[[uscheme]]** S315b⟩≡

```
case ADEF(DEFINE): return mkDef(mkDefine(out[0].name,
                                         mkLambda(out[1].names, out[2].exp)));
```

Here's how the parser might be extended

S315c. ⟨*fields of μScheme* `Component` *added in exercises* S315c⟩≡ (S313b)

```
/* if implementing COND, add a question-answer field here */
```

S315d. ⟨*arrays of shift functions added to μScheme in exercises* S315d⟩≡ (S314a)

```
/* define arrays of shift functions as needed for [[exptable]] rows */
```

S315e. ⟨*rows added to μScheme's* `exptable` *in exercises* S315e⟩≡ (S314a)

```
/* add a row for each new syntactic form of Exp */
```

S315f. ⟨*cases for μScheme's* `reduce_to_exp` *added in exercises* S315f⟩≡ (S314d)

```
/* add a case for each new syntactic form of Exp */
```

S315g. ⟨*cases for μScheme's* `reduce_to_xdef` *added in exercises* S315g⟩≡ (S315a)

```
/* add a case for each new syntactic form of definition */
```

S315h. ⟨*μScheme* `usage_table` *entries added in exercises* S315h⟩≡ (S313c)

```
/* add expected usage for each new syntactic form */
```

S315i. ⟨*shared function prototypes* S306c⟩+≡ (S303d) ◁S313d

```
void extendSyntax(void);
```

S315j. ⟨*parse.c* **[[uscheme]]** S315j⟩≡

```
void extendSyntax(void) { }
```

L.4.2 New shift functions: S-expressions and bindings

Many shift functions are reused from Impcore (Appendix G). New shift function `sSexp` calls `parsesx` to parse a literal S-expression. The result is stored in a `value` component.

S315k. ⟨*parse.c* S313c⟩+≡ ◁S315a S316a▷

```
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 `exps` fields set.

S316a. ⟨*parse.c* S313c⟩+≡ ◁S315k S316c▷

```
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.4.3 *New parsing functions: S-expressions and bindings*

Each new shift function is supported by a new parsing function.

S316b. ⟨*function prototypes for µScheme* S304a⟩+≡ (S303d) ◁S307d S323b▷

```
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.

S316c. ⟨*parse.c* S313c⟩+≡ ◁S316a S317b▷

```
Value parsesx(Par p, Sourceloc source) {
    switch (p->alt) {
    case ATOM: ⟨return p->atom interpreted as an S-expression S316d⟩
    case LIST: ⟨return p->list interpreted as an S-expression S317a⟩
    }
    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.

S316d. ⟨*return* `p->atom` *interpreted as an S-expression* S316d⟩≡ (S316c)

```
{
    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.

S317a. ⟨*return* `p->list` *interpreted as an S-expression* S317a⟩≡ (S316c)

```
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.

S317b. ⟨*parse.c* S313c⟩+≡ ◁S316c S318a▷

```
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.4.4 *Parsing atomic expressions*

To parse an atom, we need to check if it is a Boolean or integer literal. Otherwise it is a variable.

S318a. ⟨*parse.c* S313c⟩+≡ ◁S317b S326c▷

```
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.5 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.5.1 Boolean values and Boolean testing

The first part of the value interface supports Booleans.

S318b. ⟨*value.c* S318b⟩≡ S318c▷

```
bool istrue(Value v) {
    return v.alt != BOOLV || v.boolv;
}

Value truev, falsev;

void initvalue(void) {
    truev  = mkBoolv(true);
    falsev = mkBoolv(false);
}
```

L.5.2 Unspecified values

The interface defines a function to return an unspecified value. "Unspecified" means we can pick any value we like. For example, we could just always use NIL. Unfortunately, if we do that, careless persons will grow to rely on finding NIL, and they shouldn't. To foil such carelessness, we choose an unhelpful value at random.

S318c. ⟨*value.c* S318b⟩+≡ ◁S318b

```
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(-12, NULL);
        default: return mkNil();
    }
}
```

With any luck, careless persons' code might make our interpreter dereference a `NULL` pointer, which is no worse than such persons deserve.

The rest of the code deals with printing—a complex and unpleasant task.

L.5.3 *Printing and values*

The printing code is lengthy and tedious. The length and tedium are all about printing closures. When printing a closure nicely, you don't want to see the entire environment that is 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.

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`.

S319a. ⟨*printfuns.c* S319a⟩≡ S319b ▷

```
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.

S319b. ⟨*printfuns.c* S319a⟩+≡ ◁ S319a S319c ▷

```
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

S319c. ⟨*printfuns.c* S319a⟩+≡ ◁ S319b S319d ▷

```
static Namelist addfree(Name n, Namelist bound, Namelist free) {
    if (nameinlist(n, bound))
        return free;
    else
        return addname(n, free);
}
```

Here's the tree traversal. Computing the free variables of an expression is as much work as evaluating the expression. We have to know all the rules for environments.

S319d. ⟨*printfuns.c* S319a⟩+≡ ◁ S319c S321a ▷

```
Namelist freevars(Exp e, Namelist bound, Namelist free) {
    switch (e->alt) {
    case LITERAL:
```

type Exp	$\mathcal{A}$
type Explist	S303b
falsev	156b
freevars	S609i
mkLiteral	$\mathcal{A}$
mkNil	$\mathcal{A}$
mkNL	$\mathcal{A}$
mkNum	$\mathcal{A}$
mkSym	$\mathcal{A}$
mkVar	$\mathcal{A}$
type Name	43b
type Namelist	43b
nametostr	43c
type Sourceloc	S289d
strtoname	43c
synerror	48a
truev	156b
type Value	$\mathcal{A}$

```
            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 S320a⟩
            break;
        case LETX:
            ⟨let free be the free variables for e->letx S320b⟩
            break;
        ⟨extra cases for finding free variables in µScheme expressions S329c⟩
        }
        return free;
}
```

The case for lambda expressions is the interesting one. Any variables that are bound by the `lambda` are added to the "known bound" list for the recursive examination of the `lambda`'s body.

S320a. ⟨*let* `free` *be the free variables for* `e->lambdax` S320a⟩≡ (S319d)

```
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; we have to follow the rules exactly.

S320b. ⟨*let* `free` *be the free variables for* `e->letx` S320b⟩≡ (S319d)

```
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. We print a closure by printing the lambda expression, plus the values of the free variables that are not global variables. (If we included the global variables, we would be distracted by many bindings of `cons`, `car`, `+`, and so on.) Function `printnonglobals` does the hard work.

A recursive function is represented by a closure whose environment includes a pointer back to the recursive function itself. If we print such a closure by printing the values of the free variables, the printer could loop forever. The `depth` parameter cuts off this loop, so when `depth` reaches 0, the printing functions print closures simply as `<function>`.

S321a. ⟨*printfuns.c* S319a⟩+≡ ◁S319d S321b▷

```
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>");
    }
}
```

addname	S319b
bound	S319d
bprint	S188f
type Env	155a
type Explist	S303b
free	S319d
freevars	S609i
type Lambda	𝒜
type Namelist	43b
type Printbuf	S186d
printnonglobals	S322c
type Value	𝒜

The value-printing functions also need a `depth` parameter.

S321b. ⟨*printfuns.c* S319a⟩+≡ ◁S321a S322a▷

```
static void printvalueat(Printbuf output, Value v, int depth);
⟨helper functions for printvalue S322b⟩
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;
    }
}
```

If you ask just to print a value, the default depth is 0. That is, by default the interpreter doesn't print closures. If you need to debug, increase the default depth.

S322a. ⟨*printfuns.c* S319a⟩+≡ ◁S321b S322c▷

```
void printvalue(Printbuf output, va_list_box *box) {
    printvalueat(output, va_arg(box->ap, Value), 0);
}
```

Function `printtail` handles the correct printing of lists. 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.

S322b. ⟨*helper functions for* `printvalue` S322b⟩≡ (S321b)

```
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`.

S322c. ⟨*printfuns.c* S319a⟩+≡ ◁S322a S327b▷

```
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.6 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, except that instead of Impcore's separate function and value environments, the µScheme version uses the single µScheme environment.

S323a. ⟨*scheme-tests.c* S323a⟩≡ S323c ▷

```
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 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. Again, except for the environment, it's just like the Impcore version.

S323b. ⟨*function prototypes for µScheme* S304a⟩+≡ (S303d) ◁ S316b S323d ▷

```
int number_of_good_tests(UnitTestlist tests, Env rho);
```

S323c. ⟨*scheme-tests.c* S323a⟩+≡ ◁ S323a S323e ▷

```
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.

S323d. ⟨*function prototypes for µScheme* S304a⟩+≡ (S303d) ◁ S323b S325g ▷

```
TestResult test_result(UnitTest t, Env rho);
```

S323e. ⟨*scheme-tests.c* S323a⟩+≡ ◁ S323c S325h ▷

```
TestResult test_result(UnitTest t, Env rho) {
    switch (t->alt) {
    case CHECK_EXPECT:
        ⟨run check-expect test t, returning TestResult S324a⟩
    case CHECK_ASSERT:
        ⟨run check-assert test t, returning TestResult S324b⟩
    case CHECK_ERROR:
```

`bprint`	S188f
`type Env`	155a
`find`	155b
`lengthUL`	$\mathcal{A}$
`type Namelist`	43b
`type Printbuf`	S186d
`printvalueat`	S321b
`report_test_results`	S294e
`set_error_mode`	S193a
`type TestResult`	S295b
`type UnitTest`	$\mathcal{A}$
`type UnitTestlist`	S303b
`type va_list_box`	S189c
`type Value`	$\mathcal{A}$

```
            ⟨run check-error test t, returning TestResult S324c⟩
        default:
            assert(0);
        }
    }
```

Aside from the environment, there is one other difference between the µScheme `check-expect` and the Impcore `check-expect`. In Impcore, values are integers, and we test for inequality using C's `!=` operator. In µScheme, values are S-expressions, and we test for equality using C function `equalpairs` (defined below), which works the same way as the µScheme function `equal?`.

S324a. ⟨*run* `check-expect` *test* `t`, *returning* `TestResult` S324a⟩≡ (S323e)

```
{   if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_expect.check failed with an error S325b⟩
        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 S325c⟩
        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 S325a⟩
        return TEST_FAILED;
    } else {
        return TEST_PASSED;
    }
}
```

And `check-assert`

S324b. ⟨*run* `check-assert` *test* `t`, *returning* `TestResult` S324b⟩≡ (S323e)

```
{   if (setjmp(testjmp)) {
        ⟨report that evaluating t->check_assert failed with an error S325e⟩
        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 S325d⟩
        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.

S324c. ⟨*run* `check-error` *test* `t`, *returning* `TestResult` S324c⟩≡ (S323e)

```
{   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 S325f⟩
    return TEST_FAILED;
}
```

And the reporting is as in Impcore.

S325a. ⟨*report failure because the values are not equal* S325a⟩≡ (S324a)

```
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);
```

S325b. ⟨*report that evaluating* t->check_expect.check *failed with an error* S325b⟩≡ (S324a)

```
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));
```

S325c. ⟨*report that evaluating* t->check_expect.expect *failed with an error* S325c⟩≡ (S324a)

```
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));
```

S325d. ⟨*report failure because the value is false* S325d⟩≡ (S324b)

```
fprint(stderr, "Check-assert failed: %e evaluates to #f.\n", t->check_assert);
```

S325e. ⟨*report that evaluating* t->check_assert *failed with an error* S325e⟩≡ (S324b)

```
fprint(stderr, "Check-assert failed: evaluating %e causes an error: %s.\n",
               t->check_assert, bufcopy(errorbuf));
```

S325f. ⟨*report that evaluating* t->check_error *produced* check S325f⟩≡ (S324c)

```
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);
```

Function `equalpairs` tests for equality of atoms and pairs. It resembles function `equalatoms` (chunk S308a), 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`.

S325g. ⟨*function prototypes for μScheme* S304a⟩+≡ (S303d) ◁S323d S326a▷

```
bool equalpairs(Value v, Value w);
```

S325h. ⟨*scheme-tests.c* S323a⟩+≡ ◁S323e

```
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:
```


_setjmp	$\mathcal{B}$
bufreset	S186f
errorbuf	S193a
eval	157a
rho	S323e
testexp	S326a
testjmp	S193a
type Value	$\mathcal{A}$

```
            return v.sym   == w.sym;
        case NIL:
            return true;
        default:
            return false;
        }
}
```

µScheme doesn't require any change to test expressions.

S326a. ⟨*function prototypes for µScheme* S304a⟩+≡ (S303d) ◁ S325g S326d ▷

```
Exp testexp(Exp);
```

S326b. ⟨*eval.c* S326b⟩≡

```
Exp testexp(Exp e) {
    return e;
}
```

L.7 Parse-time error checking

Here is where we check for duplicate names. And LETREC for lambdas.

S326c. ⟨*parse.c* S313c⟩+≡ ◁ S318a S327a ▷

```
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,
                             "letrec tries to bind non-lambda expression %e", es->hd);
        return;
    default:
        return;
    }
}

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.8 Support for an exercise: Concatenating names

Here is an auxiliary function that will be useful if you do Exercise 54 on page 198. It concatenates names.

S326d. ⟨*function prototypes for µScheme* S304a⟩+≡ (S303d) ◁ S326a

```
Name namecat(Name n1, Name n2);
```

S327a. ⟨*parse.c* S313c⟩+≡ ◁S326c

```
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.9 Print functions for expressions

Here is the (boring) code that prints abstract-syntax trees.

S327b. ⟨*printfuns.c* S319a⟩+≡ ◁S322c S327c▷

```
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);
}
```

S327c. ⟨*printfuns.c* S319a⟩+≡ ◁S327b S328a▷

```
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);
```

bprint	S188f
type Def	$\mathcal{A}$
duplicatename	S196a
type Exp	$\mathcal{A}$
type Explist	S303b
type Name	43b
nametostr	43c
type Printbuf	S186d
type Sourceloc	S289d
strtoname	43c
synerror	48a
testexp	S340c
type va_list_box	S189c
type XDef	$\mathcal{A}$

```
}
```

S328a. ⟨*printfuns.c* S319a⟩+≡ ◁S327c S328b▷

```
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);
}
```


S328b. ⟨*printfuns.c* S319a⟩+≡ ◁S328a S329a▷

```
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 S329b⟩
        default:
            assert(0);
        }
    }
```


S329a. ⟨*printfuns.c* S319a⟩+≡ ◁S328b

```
    void printlambda(Printbuf output, va_list_box *box) {
        Lambda l = va_arg(box->ap, Lambda);
        bprint(output, "(lambda (%N) %e)", l.formals, l.body);
    }
```

L.10 Support for μScheme+

These empty definitions are placeholders for code that implements parts of μScheme+, an extension that adds control operators to μScheme. μScheme+ is the topic of Chapter 3.

S329b. ⟨*extra cases for printing μScheme ASTs* S329b⟩≡ (S328b)

S329c. ⟨*extra cases for finding free variables in μScheme expressions* S329c⟩≡ (S319d)

S329d. ⟨*lowering functions for μScheme+* S329d⟩≡ (S314a)

```
    /* placeholder */
```

L.11 Orphans

Here is a placeholder for desugarLet:

S329e. ⟨*parse.c* **[[prototype]]** S329e⟩≡

```
    Exp desugarLet(Namelist xs, Explist es, Exp body) {
        /* you replace the body of this function */
        runerror("desugaring for LET never got implemented");
        return NULL;
    }
```

bprint	S188f
type Exp	$\mathcal{A}$
type Explist	S303b
type Lambda	$\mathcal{A}$
type Namelist	43b
type Printbuf	S186d
runerror	47
type va_list_box	S189c

Chapter contents

Supporting code for μScheme+ M

M.1 Bonus exercises

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 $\langle e, \rho, \sigma, [] \rangle \rightarrow^* \langle v, \rho', \sigma', [] \rangle$.

    To prove this claim, we need a slightly stronger claim to use as an induction hypothesis:

    > 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 BIG-STEP-ASSIGN.

    (c) Prove the induction steps for derivations that end in BIG-STEP-IFTRUE or BIG-STEP-IFFALSE.

    (d) Prove the induction step for a derivation that ends in 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 BIG-STEP-WHILEEND.

    (f) Prove the induction step for a derivation that ends in BIG-STEP-WHILEITERATE.

    So far the only claim I've made formal is that if an expression e can be evaluated in μScheme, then μScheme+ evaluates e 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. When an evaluation context contains a sequence $v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n$, we represent the sequence 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 have access to a memory-analysis tool like Valgrind, what changes do you measure in the amount of allocation? 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?

M.2 Delimited continuations

The delimited-continuation primitives that best fit the semantics of this chapter 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* the continuation—instead of the stack being copied, 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.

Here are the rules:

$$\frac{}{\langle \textsc{prompt}(e), \rho, \sigma, S\rangle \rightarrow \langle e, \rho, \sigma, \textsc{prompt}(\bullet) :: S\rangle} \qquad \text{(Prompt)}$$

[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.

$$\frac{}{\langle v, \rho, \sigma, \textsc{prompt}(\bullet) :: S\rangle \to \langle v, \rho, \sigma, S\rangle} \quad (\textsc{Prompt-Finish})$$

$$\frac{}{\langle \textsc{control}(e), \rho, \sigma, S\rangle \to \langle e, \rho, \sigma, \textsc{control}(\bullet) :: S\rangle} \quad (\textsc{Control})$$

$$\frac{v_f \text{ is a function} \qquad \text{None of } F_1, \ldots, F_n \text{ has the form } \textsc{prompt}(\bullet)}{\begin{array}{l}\langle v_f, \rho, \sigma, \textsc{control}(\bullet) :: F_1 :: \cdots :: F_n :: \textsc{prompt}(\bullet) :: S\rangle \to \\ \quad \langle \textsc{apply}(v_f, \textsc{continuation}(F_1, \ldots, F_n)), \rho, \sigma, \textsc{prompt}(\bullet) :: S\rangle\end{array}} \quad (\textsc{Control-Capture})$$

$$\frac{v_f = \textsc{continuation}(F_1, \ldots, F_n)}{\langle v_1, \rho, \sigma, \textsc{apply}(v_f, \bullet) :: S\rangle \to \langle v_1, \rho, \sigma, F_1 :: \cdots :: F_n :: S\rangle} \quad (\textsc{Apply-Delimited-Continuation})$$

M.3 The evaluation stack

This section shows the implementation of the `Stack` of evaluation contexts and its instrumentation.

M.3.1 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.

S333a. ⟨*representation of* `struct Stack` S333a⟩≡ (S333b)

```
struct Stack {
    int size;
    Frame *frames;  // memory for 'size' frames
    Frame *sp;      // points to first unused frame
};
```

Instrumentation is stored in three global variables. Tail-call optimization is on by default; showing the high stack mark is not.

S333b. ⟨*context-stack.c* S333b⟩≡ S333c ▷

```
⟨representation of struct Stack S333a⟩

bool optimize_tail_calls = true;
int  high_stack_mark;   // maximum number of frames used in the current evaluation
bool show_high_stack_mark;
```

A fresh, empty stack can hold 8 frames.

S333c. ⟨*context-stack.c* S333b⟩+≡ ◁ S333b S333d ▷

```
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;
}
```

type `Frame`	225a
type `Stack`	225a

A stack that has already been allocated can be emptied by calling `clearstack`. This situation may occur if a call to `eval` is terminated prematurely (with a non-empty stack) by a call to `error`.

S333d. ⟨*context-stack.c* S333b⟩+≡ ◁ S333c S334b ▷

```
void clearstack (Stack s) {
    s->sp = s->frames;
}
```

This initialization code runs in eval and sets its local variable evalstack.

S334a. ⟨*ensure that* evalstack *is initialized and empty* S334a⟩≡ (229a)
```
if (evalstack == NULL)
    evalstack = emptystack();
else
    clearstack(evalstack);
```

Unless the sp and frames fields point to the same memory, there is a frame on top of the stack.

S334b. ⟨*context-stack.c* S333b⟩+≡ ◁S333d S334c▷
```
Frame *topframe (Stack s) {
    assert(s);
    if (s->sp == s->frames)
        return NULL;
    else
        return s->sp - 1;
}
```

S334c. ⟨*context-stack.c* S333b⟩+≡ ◁S334b S334d▷
```
Frame *topnonlabel (Stack s) {
    Frame *p;
    for (p = s->sp; p > s->frames && p[-1].form.alt == LABEL; p--)
        ;
    if (p > s->frames)
        return p-1;
    else
        return NULL;
}
```

Pushing, whether pushframe or pushenv_opt, is implemented using the private function push. Function push returns a pointer to the frame just pushed.

S334d. ⟨*context-stack.c* S333b⟩+≡ ◁S334c S334f▷
```
static Frame *push (Frame f, Stack s) {
    assert(s);
    ⟨if stack s is full, enlarge it S334e⟩
    *s->sp++ = f;
    ⟨set high_stack_mark from stack s S336d⟩
    return s->sp - 1;
}
```

Ten thousand stack frames ought to be enough for anybody.

S334e. ⟨*if stack* s *is full, enlarge it* S334e⟩≡ (S334d)
```
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;
}
```

A frame can be popped only if the stack is not empty. But there is no need for memory management or instrumentation.

S334f. ⟨*context-stack.c* S333b⟩+≡ ◁S334d S335a▷
```
void popframe (Stack s) {
    assert(s->sp - s->frames > 0);
```

```
    s->sp--;
}
```

Here's the specialized `pushframe`.

S335a. ⟨*context-stack.c* S333b⟩+≡ ◁S334f S335d▷

```
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;
}
```


M.3.2 Printing the stack

Here are the functions used to print frames and stacks. Function `printnoenv` prints the current environment as a C pointer, rather than as a list of (name, value) pairs.

S335b. ⟨*function prototypes for µScheme+* S335b⟩≡ (S346)

```
void printstack    (Printbuf, va_list_box*);
void printoneframe(Printbuf, va_list_box*);
void printframe    (Printbuf, Frame *fr);
void printnoenv    (Printbuf, va_list_box*);
```

S335c. ⟨*install printers* S335c⟩≡ (S309a)

```
installprinter('S', printstack);
installprinter('F', printoneframe);
installprinter('R', printnoenv);
```

S335d. ⟨*context-stack.c* S333b⟩+≡ ◁S335a S335e▷

```
void printnoenv(Printbuf output, va_list_box* box) {
    Env env = va_arg(box->ap, Env);
    bprint(output, "@%*", (void *)env);
}
```

S335e. ⟨*context-stack.c* S333b⟩+≡ ◁S335d S335f▷

```
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");
    }
}
```

S335f. ⟨*context-stack.c* S333b⟩+≡ ◁S335e S336a▷

```
void printoneframe(Printbuf output, va_list_box *box) {
    Frame *fr = va_arg(box->ap, Frame*);
    printframe(output, fr);
}
```

S336a. ⟨*context-stack.c* S333b⟩+≡ ◁S335f
```
void printframe (Printbuf output, Frame *fr) {
    bprint(output, "%*: ", (void *) fr);
    bprint(output, "[%e]", &fr->form);
}
```

M.3.3 Instrumentation for the high stack mark

S336b. ⟨*use the options in* `env` *to initialize the instrumentation* S336b⟩≡ (229a) S336g▷
```
high_stack_mark = 0;
show_high_stack_mark =
    istrue(getoption(strtoname("&show-high-stack-mark"), env, falsev));
```

S336c. ⟨*if* `show_high_stack_mark` *is set, show maximum stack size* S336c⟩≡ (229b)
```
if (show_high_stack_mark)
    fprintf(stderr, "High stack mark == %d\n", high_stack_mark);
```

S336d. ⟨*set* `high_stack_mark` *from stack* `s` S336d⟩≡ (S334d)
```
{   int n = s->sp - s->frames;
    if (n > high_stack_mark)
        high_stack_mark = n;
}
```

M.3.4 Tracing machine state using the stack

Variables `etick` and `vtick` count the number of state transitions involving an expression or a variable as the current item, respectively. Pointer `trace_countp` points to the value of a μScheme+ number. That way, `set` expressions in the μScheme+ code can turn tracing on and off during a single call to `eval`.

S336e. ⟨*stack-debug.c* S336e⟩≡ S336f▷
```
static int etick, vtick;  // number of times saw a current expression or value
static int *trace_countp; // if not NULL, points to value of &trace-stack
```

Initalization sets the private variables.

S336f. ⟨*stack-debug.c* S336e⟩+≡ ◁S336e S337a▷
```
void stack_trace_init(int *countp) {
    etick = vtick = 0;
    trace_countp = countp;
}
```

The following code runs in `eval`, which has access to `env`. There's just a little sanity checking—if someone changes μScheme+ variable `&trace-stack` from a number to a non-number, chaos may ensue.

S336g. ⟨*use the options in* `env` *to initialize the instrumentation* S336b⟩+≡ (229a) ◁S336b S342c▷
```
{   Value *p = find(strtoname("&trace-stack"), env);
    if (p && p->alt == NUM)
        stack_trace_init(&p->num);
    else
        stack_trace_init(NULL);
}
```

Tracing a current expression shows the tick number, the expression, a pointer to the environment, and the stack. The trace count is decremented.

S337a. ⟨*stack-debug.c* S336e⟩+≡ ◁S336f S337b▷

```
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);
    }
}
```


Tracing a current value works the same way, except I use a special rendering for the empty stack.

S337b. ⟨*stack-debug.c* S336e⟩+≡ ◁S337a

```
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.4 Updating lists of expressions within contexts

Section 3.6.8 describes several functions I use to implement the evaluation of APPLY, LET, and other forms that use an `Explist` to remember a list of values.

S337c. ⟨*context-lists.c* S337c⟩≡ S337d▷

⟨*private functions for updating lists of expressions in contexts* S337e⟩

To move hole from one position to the next, I find the hole, fill it, and then place a hole at the beginning of the rest of the list.

S337d. ⟨*context-lists.c* S337c⟩+≡ ◁S337c S338b▷

```
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);
}
```

S337e. ⟨*private functions for updating lists of expressions in contexts* S337e⟩≡ (S337c) S338a▷

```
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`.

S338a. ⟨*private functions for updating lists of expressions in contexts* S337e⟩+≡ (S337c) ◁S337e

```
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.

S338b. ⟨*context-lists.c* S337c⟩+≡ ◁S337d S338c▷

```
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.

S338c. ⟨*context-lists.c* S337c⟩+≡ ◁S338b S338d▷

```
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.

S338d. ⟨*context-lists.c* S337c⟩+≡ ◁S338c S338e▷

```
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.

S338e. ⟨*context-lists.c* S337c⟩+≡ ◁S338d S339a▷

```
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.

S339a. ⟨*context-lists.c* S337c⟩+≡ ◁S338e S339b▷

```
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.

S339b. ⟨*context-lists.c* S337c⟩+≡ ◁S339a

```
Value asLiteral(Exp e) {
  assert(e->alt == LITERAL);
  return validate(e->literal);
}
```

M.5 Lowering

S339c. ⟨*lower.c* S339c⟩≡ S339d▷

```
#define LOWER_RETURN false // to do return-lowering exercise, change me
```

S339d. ⟨*lower.c* S339c⟩+≡ ◁S339c S339e▷

```
static inline Exp lowerLet1(Name x, Exp e, Exp body) {
    return mkLetx(LET, mkNL(x, NULL), mkEL(e, NULL), body);
}
```

S339e. ⟨*lower.c* S339c⟩+≡ ◁S339d S339f▷

```
static Exp lowerSequence(Exp e1, Exp e2) {
    return lowerLet1(strtoname("ignore me"), e1, e2);
}
```

S339f. ⟨*lower.c* S339c⟩+≡ ◁S339e S339g▷

```
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));
}
```

S339g. ⟨*lower.c* S339c⟩+≡ ◁S339f S339h▷

```
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);
    }
}
```

S339h. ⟨*lower.c* S339c⟩+≡ ◁S339g S340a▷

```
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));
    }
}
```

S340a. ⟨*lower.c* S339c⟩+≡ ◁S339h S340b▷

```
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,
                                   mkLabel(strtoname(":return"), body));
        d->define.lambda.body = body;
        break;
    }
    default:     assert(0);
    }
}
```


We can't lower a test eagerly, because if lowering fails with an error, it has to occur in the right dynamic context.

S340b. ⟨*lower.c* S339c⟩+≡ ◁S340a S340c▷

```
void lowerXdef(XDef d) {
    switch (d->alt) {
    case DEF: lowerDef(d->def); break;
    case USE: break;
    case TEST: break;
    default: assert(0);
    }
}
```

S340c. ⟨*lower.c* S339c⟩+≡ ◁S340b S340d▷

```
Exp testexp(Exp e) {
    return lower(0, e);
}
```

S340d. ⟨*lower.c* S339c⟩+≡ ◁S340c

⟨*definition of private function* `lower` 228e⟩

S340e. ⟨*other cases for lowering expression* `e` S340e⟩≡ (228e)

```
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;
case WHILEX: {
    LoweringContext nc = c | LOOPCONTEXT;
    Exp body = mkLabel(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       = mkLabel(strtoname(":break"), loop);
    return mkLowered(e, lowered);
```

```
}
case BEGIN:
    lowerAll(c, e->begin);
    return mkLowered(e, lowerBegin(e->begin));
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);
    }
case LAMBDAX: {
    LoweringContext nc = FUNCONTEXT;  // no loop!
    Exp body    = lower(nc, e->lambdax.body);
    e->lambdax.body =
        LOWER_RETURN ? mkLowered(e->lambdax.body, mkLabel(strtoname(":return"), body))
                     : body;
    return e;
}
case APPLY:
    lowerAll(c, e->apply.actuals);
    e->apply.fn = lower(c, e->apply.fn);
    return e;
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->r
                            : e;
    else
        othererror("Lowering error: %e appeared outside of any function", e);
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 = mkLabel(e->try_catch.label, lbody);
    Exp lowered = lowerLet1(h, handler, mkApply(labeled, mkEL(mkVar(h), NULL)));
    return mkLowered(e, lowered);
}
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));
```

type Def 𝒜
type Exp 𝒜
falsev 156b
type Lambda 𝒜
lower S339g
lowerAll S339g
lowerBegin S339f
type LoweringContext 228d
lowerLet1 S339d
lowerLetstar S339h
lowerSequence S339e
mkIfx 𝒜
mkLabel 𝒜
mkLambdax 𝒜
mkLiteral 𝒜
mkLongGoto 𝒜
mkLoopback 𝒜
mkLowered 𝒜
type Name 43b
othererror S195b
strtoname 43c
type XDef 𝒜

```
        Exp lowered = lowerLet1(x, lower(c, e->throw.exp), throw);
        return mkLowered(e, lowered);
    }
    case LABEL:
        e->label.body = lower(c, e->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);
```

S342a. ⟨*lower definition* `d` *as needed* S342a⟩≡ (S305e)

```
lowerXdef(d);
```

M.6 Options and diagnostic code

S342b. ⟨*options.c* S342b⟩≡

```
Value getoption(Name name, Env env, Value defaultval) {
    Value *p = find(name, env);
    if (p)
        return *p;
    else
        return defaultval;
}
```

S342c. ⟨*use the options in* `env` *to initialize the instrumentation* S336b⟩+≡ (229a) ◁S336g

```
optimize_tail_calls =
    istrue(getoption(strtoname("&optimize-tail-calls"), env, truev));
```

S342d. ⟨*validate.c* S342d⟩≡

```
Value validate(Value v) {
    return v;
}
```

S342e. ⟨*cases for forms that appear only as frames* S342e⟩≡ (230a)

```
case HOLE:
case ENV:
    assert(0);
```

S342f. ⟨*definition of static* `Exp hole`*, which always has a hole* S342f⟩≡ (229a)

```
static struct Exp holeExp = { HOLE, { { NIL, { 0 } } } };
static Exp hole = &holeExp;
```

M.7 Parsing

S342g. ⟨*arrays of shift functions added to μScheme in exercises* S342g⟩≡ (S314a) S343b ▷

```
ShiftFun breakshifts[]  = { stop };
ShiftFun returnshifts[] = { sExp, stop };
ShiftFun throwshifts[]  = { sName, sExp, stop };
ShiftFun tcshifts[]     = { sExp, sName, sExp, stop };
ShiftFun labelshifts[]  = { sName, sExp, stop };
```

S343a. ⟨*rows added to μScheme's* `exptable` *in exercises* S343a⟩≡ (S314a)
```
{ "break",     BREAKX,        breakshifts },
{ "continue",  CONTINUEX,     breakshifts },
{ "return",    RETURNX,       returnshifts },
{ "throw",     THROW,         throwshifts },
{ "try-catch", TRY_CATCH,     tcshifts },
{ "label",     LABEL,         labelshifts },
{ "long-goto", LONG_GOTO,     labelshifts },
{ "when",      SUGAR(WHEN),   applyshifts },
{ "unless",    SUGAR(UNLESS), applyshifts },
```


OK, not really exercises...

S343b. ⟨*arrays of shift functions added to μScheme in exercises* S342g⟩+≡ (S314a) ◁ S342g
```
static ShiftFun procwhileshifts[] = { sExp, sExps,      stop };
static ShiftFun procletshifts[]   = { sBindings, sExps, stop };
```

S343c. ⟨*rows of μScheme's* `exptable` *that are sugared in μScheme+* **[[uschemeplus]]** S343c⟩≡
```
  { "while",  ANEXP(WHILEX),  procwhileshifts },
  { "let",    ALET(LET),      procletshifts },
  { "let*",   ALET(LETSTAR),  procletshifts },
  { "letrec", ALET(LETREC),   procletshifts },
```

S343d. ⟨*cases for μScheme's* `reduce_to_exp` *added in exercises* S343d⟩≡ (S314d)
```
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].e
case ANEXP(LABEL):     return mkLabel(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));
```

S343e. ⟨*cases for* `reduce_to_exp` *that are sugared in μScheme+* **[[uschemeplus]]** S343e⟩≡
```
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));
```

S343f. ⟨*μScheme* `usage_table` *entries added in exercises* S343f⟩≡ (S313c)
```
{ 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(LABEL),     "(label lbl-name body)" },
{ ANEXP(LONG_GOTO), "(long-goto lbl-name exp)" },
```

S343g. ⟨*lowering functions for μScheme+* S343g⟩≡ (S314a)
```
static Exp smartBegin(Explist es) {
    if (es != NULL && es->tl == NULL)
        return es->hd;
    else
        return mkBegin(es);
}
```

`code`, in μScheme (in GC?!)	S360b
in μScheme+	S314d
`comps`, in μScheme (in GC?!)	S360b
in μScheme+	S314d
`type Env`	155a
`type Exp`	$\mathcal{A}$
`type Explist`	S303b
`falsev`	156b
`find`	155b
`letshifts`	S314a
`lowerXdef`	228f
`mkBegin`	$\mathcal{A}$
`mkBreakx`	$\mathcal{A}$
`mkContinuex`	$\mathcal{A}$
`mkIfx`	$\mathcal{A}$
`mkLabel`	$\mathcal{A}$
`mkLetx`	$\mathcal{A}$
`mkLiteral`	$\mathcal{A}$
`mkLongGoto`	$\mathcal{A}$
`mkReturnx`	$\mathcal{A}$
`mkThrow`	$\mathcal{A}$
`mkTryCatch`	$\mathcal{A}$
`mkWhilex`	$\mathcal{A}$
`type Name`	43b
`sBindings`	S313d
`sExp`	S207e
`sExps`	S207e
`type ShiftFun`	S207d
`sName`	S207e
`stop`	S209d
`type Value`	$\mathcal{A}$
`whileshifts`	S314a

S344a. ⟨`reduce_to_xdef` *case for* `ADEF(DEFINE)` **[[uschemeplus]]** S344a⟩≡

```
case ADEF(DEFINE):
  return mkDef(mkDefine(out[0].name,
                        mkLambda(out[1].names, smartBegin(out[2].exps))));
```

S344b. ⟨*extend-syntax.c* S344b⟩≡

```
extern void extendDefine(void);
void extendSyntax(void) { extendDefine(); }
```

S344c. ⟨*extra cases for printing μScheme ASTs* S344c⟩≡ (S328b)

```
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_ca
    break;
case LABEL:
    bprint(output, "(label %n %e)", e->label.label, e->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;
case LOWERED:
    bprint(output, "%e", e->lowered.before);
    break;
case LOOPBACK:
    bprint(output, "...loopback...");
    break;
```

M.8 Finding free variables

Here are extra cases for the `freevars` function, which is used to do a good job printing closures.

S344d. ⟨*extra cases for finding free variables in μScheme expressions* S344d⟩≡ (S319d) S345a ▷

```
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;
    case LABEL:
        free = freevars(e->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;
```


These forms appear only in contexts, and we have no business looking for a free variable.

S345a. ⟨*extra cases for finding free variables in μScheme expressions* S344d⟩+≡ (S319d) ◁S344d

```
    case HOLE:
    case ENV:
        assert(0);
        break;
```

M.9 Interpreter code omitted from the chapter

S345b. ⟨*cases for forms that never appear as frames* S345b⟩≡ (230b)

```
    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);
```

S345c. ⟨*bind every name in* `e->letx.xs` *to an unspecified value in* `env` S345c⟩≡ (236d)

```
    {   Namelist xs;
        for (xs = e->letx.xs; xs; xs = xs->tl)
            env = bindalloc(xs->hd, unspecified(), env);
    }
```

S345d. ⟨*if not all of* `e->letx.es` *are lambdas, reject the* `letrec` S345d⟩≡ (236d)

```
    for (Explist es = e->letx.es; es; es = es->tl)
        if (es->hd->alt != LAMBDAX)
            runerror("letrec tries to bind non-lambda expression %e", es->hd);
```

`bindalloc`	155c
`bound`	S319d
`bprint`	S188f
`env`	229a
`type Explist`	S303b
`extendDefine`	S213c
`free`	S319d
`freevars`	S609i
`mkDef`	$\mathcal{A}$
`mkDefine`	$\mathcal{A}$
`mkLambda`	$\mathcal{A}$
`type Namelist`	43b
`out`	S315a
`output`	S328b
`runerror`	47
`smartBegin`	S343g
`unspecified`	156d

M.10 BUREAUCRACY

As in μScheme, we gather all the interfaces into a single C header file.

S346. ⟨all.h *for μScheme+* S346⟩≡

```
#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* S303c⟩
⟨*type definitions for μScheme+* 225a⟩
⟨*type definitions for μScheme* 147b⟩
⟨*shared type definitions* 43b⟩

⟨*structure definitions for μScheme+* 225b⟩
⟨*structure definitions for μScheme* S313b⟩
⟨*shared structure definitions* generated automatically⟩

⟨*function prototypes for μScheme+* S335b⟩
⟨*function prototypes for μScheme* 155b⟩
⟨*shared function prototypes* 43c⟩

⟨*global variables for μScheme+* 226d⟩

⟨*macro definitions used in parsing* generated automatically⟩
⟨*declarations of global variables used in lexical analysis and parsing* generated automatically⟩

Chapter contents

Supporting code for garbage collection

N

This appendix shows supporting code that can help with the Exercises in Chapter 4: visiting functions, scanning procedures, root-tracking code for the evaluator, and the implementation of the root stack.

N.1 Bureaucracy

Data structures for roots.

S349a. ⟨*type definitions for μScheme+* S349a⟩≡ (S346)
```
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.

S349b. ⟨*structure definitions for μScheme+* S349b⟩≡ (S346)
⟨*structure definitions used in garbage collection* 269a⟩

S349c. ⟨*global variables for μScheme+* S349c⟩≡ (S346)
⟨*global variables used in garbage collection* S356c⟩

These are the scan calls.

S349d. ⟨*scan frame* `*fr`*, forwarding all internal pointers* S349d⟩≡ (278)
```
scanframe(fr);
```

S349e. ⟨*scan list of unit tests* `testss->hd`*, forwarding all internal pointers* S349e⟩≡ (278)
```
scantests(testss->hd);
```

S349f. ⟨*scan register* `regs->hd`*, forwarding all internal pointers* S349f⟩≡ (278)
```
scanloc(regs->hd);
```

S349g. ⟨*scan object* `*scanp`*, forwarding all internal pointers* S349g⟩≡ (278)
```
scanloc(scanp);
```

N.2 Basic support for the two collectors

N.2.1 Object-visiting procedures for mark-and-sweep collection

Section 4.4.2 presents a few procedures for visiting μScheme objects in a depth-first search. The remaining procedures are here.

To visit an expression, we visit its literal value, if any, and of course its subexpressions.

S349h. ⟨*ms.c* S349h⟩≡ S351b ▷
```
static void visitexp(Exp e) {
    switch (e->alt) {
    ⟨cases for visitexp S350a⟩
    }
    assert(0);
}
```

There are more cases than will fit on a page, so I break them into three groups. First, μScheme expressions:

S350a. ⟨*cases for* `visitexp` S350a⟩≡ (S349h) S350b ▷

```
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;
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, μScheme+ expressions:

S350b. ⟨*cases for* `visitexp` S350a⟩+≡ (S349h) ◁ S350a S351a ▷

```
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 LABEL:
    visitexp(e->label.body);
    return;
case LOWERED:
```

```
        visitexp(e->lowered.before);
        return;
    case LOOPBACK:
        return;
```

Last, μScheme+ evaluation contexts:

S351a. ⟨*cases for* `visitexp` S350a⟩+≡ (S349h) ◁S350b

```
    case ENV:
        visitenv(e->env.contents);
        return;
    case HOLE:
        return;
```

Function `visitexplist` visits a list of expressions.

S351b. ⟨*ms.c* S349h⟩+≡ ◁S349h S351c▷

```
static void visitexplist(Explist es) {
    for (; es; es = es->tl)
        visitexp(es->hd);
}
```

Function `visitregiserlist` visits a list of registers.

S351c. ⟨*ms.c* S349h⟩+≡ ◁S351b S351d▷

```
static void visitregisterlist(Registerlist regs) {
    for ( ; regs != NULL; regs = regs->tl)
        visitregister(regs->hd);
}
```

To visit a `Stack`, we have to be able to see the representation. Then we visit all the frames.

S351d. ⟨*ms.c* S349h⟩+≡ ◁S351c S351e▷

```
⟨representation of struct Stack S333a⟩
static void visitstack(Stack s) {
    Frame *fr;
    for (fr = s->frames; fr < s->sp; fr++) {
        visitframe(fr);
    }
}
```

Visiting a frame means visiting both expressions.

S351e. ⟨*ms.c* S349h⟩+≡ ◁S351d S351f▷

```
static void visitframe(Frame *fr) {
    visitexp(&fr->form);
    if (fr->syntax != NULL)
        visitexp(fr->syntax);
}
```

Visiting lists of pending unit tests visits all lists on the list.

S351f. ⟨*ms.c* S349h⟩+≡ ◁S351e S352a▷

```
static void visittestlists(UnitTestlistlist uss) {
    UnitTestlist ul;

    for ( ; uss != NULL; uss = uss->tl)
        for (ul = uss->hd; ul; ul = ul->tl)
            visittest(ul->hd);
}
```


type `Explist`	S303b
type `Frame`	225a
type `Registerlist`	S349a
type `Stack`	225a
type `UnitTestlist`	S303b
type `UnitTestlistlist`	S349a
`visitenv`	273b
`visitexp`	273b
`visitexplist`	273b
`visitframe`	273b
`visitvalue`	273b

Visiting a unit test means visiting its component expressions.

S352a. ⟨*ms.c* S349h⟩+≡ ◁S351f S352b▷

```
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.

S352b. ⟨*ms.c* S349h⟩+≡ ◁S352a

```
static void visitroots(void) {
    visitenv(*roots.globals.user);
    visittestlists(roots.globals.internal.pending_tests);
    visitstack(roots.stack);
    visitregisterlist(roots.registers);
}
```

N.2.2 Root-scanning procedures for copying collection

Section 4.5.3 presents a few procedures for scanning potential roots. The rest are here. As explained in Section 4.5.3, 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 chunk 282b. 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.

Scanning expressions means scanning internal values or subexpressions.

S352c. ⟨*copy.c* S352c⟩≡ S354a▷

```
static void scanexp(Exp e) {
    switch (e->alt) {
    ⟨cases for scanexp S352d⟩
    }
    assert(0);
}
```

First, μScheme expressions:

S352d. ⟨*cases for* `scanexp` S352d⟩≡ (S352c) S353a▷

```
case LITERAL:
    scanloc(&e->literal);
    return;
case VAR:
    return;
case IFX:
    scanexp(e->ifx.cond);
    scanexp(e->ifx.truex);
```

```
        scanexp(e->ifx.falsex);
        return;
    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:

S353a. ⟨*cases for* `scanexp` S352d⟩+≡ (S352c) ◁S352d S353b▷

```
    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 LABEL:
        scanexp(e->label.body);
        return;
    case LOWERED:
        scanexp(e->lowered.before);
        scanexp(e->lowered.after);
        return;
    case LOOPBACK:
        return;
```

`type Exp`	$\mathcal{A}$
`roots`	269b
`scanenv`	281d
`scanexp`	281d
`scanexplist`	281d
`scanloc`	281d
`type UnitTest`	$\mathcal{A}$
`visitenv`	273b
`visitexp`	273b
`visitregisterlist`	273b
`visitstack`	273b
`visittestlists`	273b

Last, μScheme+ evaluation contexts.

S353b. ⟨*cases for* `scanexp` S352d⟩+≡ (S352c) ◁S353a

```
    case HOLE:
        return;
    case ENV:
        scanenv(e->env.contents);
```

```
        return;
```

Scanning a frame means scanning its expressions.

S354a. ⟨*copy.c* S352c⟩+≡ ◁S352c S354b▷

```
static void scanframe(Frame *fr) {
    scanexp(&fr->form);
    if (fr->syntax != NULL)
        scanexp(fr->syntax);
}
```


Function `scanexplist` scans a list of expressions.

S354b. ⟨*copy.c* S352c⟩+≡ ◁S354a S354c▷

```
static void scanexplist(Explist es) {
    for (; es; es = es->tl)
        scanexp(es->hd);
}
```

Scanning a source means scanning its pending tests.

S354c. ⟨*copy.c* S352c⟩+≡ ◁S354b S354d▷

```
static void scantests(UnitTestlist tests) {
    for (; tests; tests = tests->tl)
        scantest(tests->hd);
}
```

Scanning a test means scanning its expressions.

S354d. ⟨*copy.c* S352c⟩+≡ ◁S354c

```
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.2.3 Access to the desired size of the heap

To control the size of the heap, we might want to use the μScheme variable `&gamma-desired`, as described in Exercises 10 and 3. This routine gets the value of that variable.

S354e. ⟨*loc.c* S354e⟩≡ S357f▷

```
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;
}
```

S354f. ⟨*function prototypes for μScheme* S354f⟩≡ (S303d S346)

```
int gammadesired(int defaultval, int minimum);
```

N.2.4 *Code to push and pop register roots*

The `roots` data structure is defined here.

S355a. ⟨*root.c* S355a⟩≡ S355b ▷
```
struct Roots roots = { { NULL, { NULL } }, NULL, NULL };
```

Here are implementations of `pushreg` and `popreg`.

S355b. ⟨*root.c* S355a⟩+≡ ◁ S355a S355c ▷
```
void pushreg(Value *reg) {
    roots.registers = mkRL(reg, roots.registers);
}
```

Popping a register requires a check that the roots match.

S355c. ⟨*root.c* S355a⟩+≡ ◁ S355b S355d ▷
```
void popreg(Value *reg) {
    Registerlist regs = roots.registers;
    assert(regs != NULL);
    assert(reg == regs->hd);
    roots.registers = regs->tl;
    free(regs);
}
```

When pushing and popping a list of registers, we push left to right and pop right to left.

S355d. ⟨*root.c* S355a⟩+≡ ◁ S355c
```
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);
    }
}
```

N.3 GC DEBUGGING, WITH OR WITHOUT VALGRIND

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, we tell Valgrind 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 4.9 on page 292.)
- When memory is given from the collector to the interpreter, we tell Valgrind that it is OK to write but not OK to read until it has been initialized.

type Explist	S303b
find	155b
type Frame	225a
mkRL	$\mathcal{A}$
popreg	270a
popregs	270b
pushreg	270a
type Registerlist	S349a
roots	269b
scanexp	281d
scantest	281d
strtoname	43c
type UnitTest	$\mathcal{A}$
type UnitTestlist	S303b
type Value	$\mathcal{A}$
type Valuelist	S303c

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.

S356a. ⟨*gcdebug.c* S356a⟩≡ S356d ▷

```
#ifndef NOVALGRIND
  #include <valgrind/memcheck.h>
#else
  ⟨define do-nothing replacements for Valgrind macros S356b⟩
#endif
```


To prevent compiler warnings, the do-nothing macros "evaluate" their arguments by casting them to `void`.

S356b. ⟨*define do-nothing replacements for Valgrind macros* S356b⟩≡ (S356a)

```
#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))
```

The Valgrind calls are described in Valgrind's documentation for "custom memory allocators."

At initialization we create 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. We also initialize the `gcverbose` flag.

S356c. ⟨*global variables used in garbage collection* S356c⟩≡ (S349c)

```
extern bool gc_uses_mark_bits;
```

S356d. ⟨*gcdebug.c* S356a⟩+≡ ◁ S356a S356e ▷

```
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 we acquire objects, we make each one invalid, we tell Valgrind that each one exists, and we mark all the memory as inaccessible (because it belongs to the collector).

S356e. ⟨*gcdebug.c* S356a⟩+≡ ◁ S356d S356f ▷

```
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 S357d⟩
    VALGRIND_MAKE_MEM_NOACCESS(mem, nvalues * sizeof(*mem));
}
```

Before we release memory, we check that the objects are invalid. We have to tell Valgrind that it's temporarily OK to look at the object.

S356f. ⟨*gcdebug.c* S356a⟩+≡ ◁ S356e S357a ▷

```
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 handing an object to the interpreter, we tell Valgrind it's been allocated, we make it invalid, and finally tell Valgrind that it's writable but uninitialized.

S357a. ⟨*gcdebug.c* S356a⟩+≡ ◁S356f S357b▷

```
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));
}
```

When we get an object back, we check that it's *not* invalid (because it should have been initialized to a valid value immediately after it was allocated). Then we mark it invalid and tell Valgrind it's been freed.

S357b. ⟨*gcdebug.c* S356a⟩+≡ ◁S357a S357c▷

```
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);
}
```

The loop to reclaim a block works only if the pointer is a pointer to an array of `Value`, not an array of `Mvalue`.

S357c. ⟨*gcdebug.c* S356a⟩+≡ ◁S357b S358a▷

```
void gc_debug_post_reclaim_block(Value *mem, unsigned nvalues) {
    unsigned i;
    ⟨when using mark bits, barf unless nvalues is 1 S357d⟩
    for (i = 0; i < nvalues; i++)
        gc_debug_post_reclaim(&mem[i]);
}
```

S357d. ⟨*when using mark bits, barf unless* `nvalues` *is 1* S357d⟩≡ (S356e 357c)

```
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.

S357e. ⟨*validate.c* S357e⟩≡

```
Value validate(Value v) {
    assert(v.alt != INVALID);
    return v;
}
```

Collector initialization uses the ANSI C function `atexit` to make sure that before the program exits, final garbage-collection statistics are printed.

S357f. ⟨*loc.c* S354e⟩+≡ ◁S354e

```
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);
}
```


Here are the printing functions.

S358a. ⟨*gcdebug.c* S356a⟩+≡ ◁S357c S358b▷

```
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);
  }
}
```

S358b. ⟨*gcdebug.c* S356a⟩+≡ ◁S358a S361c▷

```
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.4 Code that is changed to support garbage collection

Most parts of the μScheme+ interpreter are either replaced completely or used without change. But a few are modified versions of the originals. The modifications have to do with keeping track of the root set: they are codes than can allocate, and the modifications make sure that before `allocloc` can be called, the root set is up to date. To keep the root set up to date, I frequently abuse the stack of evaluation contexts. If I need to save an `Exp` or an `Env`, for example, I push an appropriate context. Because I pop the context immediately afterward, the evaluator never sees these abusive contexts, and they don't interfere with evaluation. (If I need to save a `Value`, on the other hand, I simply use `pushreg` or `pushregs` as intended.)

Code that is modified or added to support garbage collection is shown in *`typewriter italics`*.

N.4.1 Revised environment-extension routines

To be sure that the current environment is always visible to the garbage collector, we need 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. We put it on the stack so that when allocate is called, the bindings in env are kept live.

S359a. ⟨*env.c* S359a⟩≡ S359b ▷

```
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 we have a fresh copy of it. It contains Value* pointers, so 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 we get to allocate, our copy of val is dead—only allocate's private copy matters.

In bindalloclist, by contrast, when we call bindalloc with vs->hd, our copy of vs->hd is dead, as is everything that precedes it. But values reachable from vs->tl are still live. To make them visible to the garbage collector, we treat them as "machine registers."

S359b. ⟨*env.c* S359a⟩+≡ ◁ S359a

```
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;
}
```

allocate	156a
bindalloc	155c
type Env	155a
evalstack	229a
freebuf	S186e
gcverbose	S356d
mkEnvStruct	$\mathcal{A}$
type Name	43b
type Namelist	43b
popframe	226a
popregs	270b
type Printbuf	S186d
printbuf	S186e
pushframe	226a
pushregs	270b
roots	269b
type va_list_box	S189c
type Value	$\mathcal{A}$
type Valuelist	S303c
vbprint	S189d

N.4.2 Revisions to eval

Chapter 3's eval function needs just a couple of changes to support garbage collection. First, the evaluation stack is part of the root set:

S359c. ⟨*ensure that* evalstack *is initialized and empty* S359c⟩≡ (229a)

```
assert(topframe(roots.stack) == NULL);
roots.stack = evalstack;
```

Second, the primitive cons can allocate. So the local variable env has to be made visible to the garbage collector. I just put it on the stack.

S359d. ⟨*apply* fn.primitive *to* vs *and transition to the next state* S359d⟩≡

N.4.3 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 we push it onto the context stack. And because the garbage collector might move objects, after allocating, we overwrite the original right-hand side with the version from the top of the stack.

S360a. ⟨*evaluate* `val` *binding and return new environment* S360a⟩≡ (161e)

```
{
    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 S305c⟩
    return env;
}
```


N.4.4 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`, we must make sure that any quoted S-expression is visible as a root. Again, I make them visible by abusing the stack of evaluation contexts: in `reduce_to_exp`, if I see a quoted S-expression, I put in on the stack. Since it's the `Exp` we want on the stack, not just the `struct Exp`, I wrap it in a BEGIN expression:

S360b. ⟨*parse.c* S360b⟩≡

```
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+ generated automatically⟩
    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 S315f⟩
    }
    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 S317a, because there's no garbage collector in play, we simply call `parsesx` on the `hd` and `tl` and then call `cons` on the result. With a garbage collector, this simple code won't work: if the second call triggers a garbage collection, the result of the first call has to be a root. So the first result goes (temporarily) into a "machine register."

S361a. ⟨*return* `p->list` *interpreted as an S-expression* S361a⟩≡ (S316c)
```
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.4.5 Checking for cycles in `cons`

I've left in this early-stage debugging code, which looks for a cycle after every `cons`.

S361b. ⟨*function prototypes for μScheme+* S361b⟩≡ (S346)
```
void cyclecheck(Value *l);
```

The code uses depth-first search to make sure no value is ever its own ancestor.

S361c. ⟨*gcdebug.c* S356a⟩+≡ ◁S358b S361d▷
```
struct va { /* value ancestors */
    Value *l;
    struct va *parent;
};
```

S361d. ⟨*gcdebug.c* S356a⟩+≡ ◁S361c S361e▷
```
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");
        }
}
```

S361e. ⟨*gcdebug.c* S356a⟩+≡ ◁S361d
```
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.5 Placeholders for exercises

S362a. ⟨*private declarations for copying collection* S362a⟩≡

```
static void collect(void);
```

S362b. ⟨*copy.c* **[[prototype]]** S362b⟩≡

```
/* 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;
```

S362c. ⟨*ms.c* **[[prototype]]** S362c⟩≡ S362d ▷

```
/* you need to redefine these functions */
void printfinalstats(void) {
  (void)nalloc; (void)ncollections; (void)nmarks;
  assert(0);
}
```

S362d. ⟨*ms.c* **[[prototype]]** S362c⟩+≡ ◁ S362c

```
void avoid_unpleasant_compiler_warnings(void) {
    (void)visitroots;
}
```

Chapter contents

Supporting code for the ML interpreter for μScheme

O

This appendix describes language-specific code that is used to implement μScheme but is not interesting enough to include in Chapter 5. This code includes code for lexical analysis, for parsing, and for running unit tests, as does a similar appendix for every bridge language that is implemented in ML. The code for μScheme also includes an implementation of the "unspecified" values in the operational semantics.

O.1 Interpreter infrastructure

The code in this section is a late addition to the Supplement. Some of it ought to migrate into Appendix I.

Extended definitions

S365a. ⟨*definition of* `unit_test` *for untyped languages (shared)* S365a⟩≡ (S365c)

```
datatype unit_test = CHECK_EXPECT of exp * exp
                   | CHECK_ASSERT of exp
                   | CHECK_ERROR  of exp
```

S365b. ⟨*definition of* `xdef` *(shared)* S365b⟩≡ (S365c)

```
datatype xdef = DEF    of def
              | USE    of name
              | TEST   of unit_test
```

All these type definitions, together with definitions of functions `valueString` and `expString`, are pulled together in one Noweb code chunk labeled ⟨*abstract syntax and values for μScheme* S365c⟩.

S365c. ⟨*abstract syntax and values for μScheme* S365c⟩≡ (S373a)

```
valueString      : value -> string
expString        : exp   -> string
```

⟨*definitions of* `exp` *and* `value` *for μScheme* 313a⟩
⟨*definition of* `def` *for μScheme* 313b⟩
⟨*definition of* `unit_test` *for untyped languages (shared)* S365a⟩
⟨*definition of* `xdef` *(shared)* S365b⟩
⟨*definition of* `valueString` *for μScheme, Typed μScheme, and nano-ML* 314⟩
⟨*definition of* `expString` *for μScheme* S378c⟩

Operations on values

Equality The interpreter uses equality in two places: in the = primitive and in the `check-expect` unit test. The primitive version permits only atoms to be considered equal.

S365d. ⟨*utility functions on μScheme, Typed μScheme, and nano-ML values* S365d⟩≡ (S373a) S366a ▷

```
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.

S366a. ⟨*utility functions on* μ*Scheme, Typed* μ*Scheme, and nano-ML values* S365d⟩+≡ (S373a) ◁S365d S366b▷

`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 `testEqual`.

S366b. ⟨*utility functions on* μ*Scheme, Typed* μ*Scheme, and nano-ML values* S365d⟩+≡ (S373a) ◁S366a S379▷

`testEqual : value * value -> bool`

```
val testEqual = equalpairs
```

O.1.1 *Error detection and signaling*

Every run-time error is signaled by raising the `RuntimeError` exception, which carries an error message.

S366c. ⟨*support for detecting and signaling errors detected at run time* S366c⟩≡ (S237a) S366e▷

```
exception RuntimeError of string (* error message *)
```

As in Chapter 2, duplicate names are treated as run-time errors. If a name x occurs more than twice on a list, function `duplicatename` returns `SOME` x; otherwise it returns `NONE`.

S366d. ⟨*support for names and environments* S366d⟩≡ (S237a)

`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
```

Function `errorIfDups` raises the exception if a duplicate name is found. Parameter `what` says what kind of name we're looking at, and `context` says in what context.

S366e. ⟨*support for detecting and signaling errors detected at run time* S366c⟩+≡ (S237a) ◁S366c S366f▷

`errorIfDups : string * name list * string -> unit`

```
fun errorIfDups (what, xs, context) =
  case duplicatename xs
    of NONE   => ()
     | SOME x => raise RuntimeError (what ^ " " ^ x ^ " appears twice in " ^ context)
```

Some errors might be caused not by a fault in μScheme code but in my implementation of μScheme. For those times, there's the `InternalError` exception.

S366f. ⟨*support for detecting and signaling errors detected at run time* S366c⟩+≡ (S237a) ◁S366e

```
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.

O.1.2 Extra checking for `letrec`

S367a. ⟨*if any expression in* `values` *is not a* `lambda`*, reject the* `letrec` S367a⟩≡ (318a)

```
fun insistLambda (LAMBDA _) = ()
  | insistLambda e =
      raise RuntimeError ("letrec tries to bind non-lambda expression " ^
                          expString e)
val _ = app insistLambda values
```

O.1.3 Primitives

More type predicates.

S367b. ⟨*primitives for μScheme* :: S367b⟩≡ (S372a) S367c ▷

```
("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:

S367c. ⟨*primitives for μScheme* :: S367b⟩+≡ (S372a) ◁ S367b S367d ▷

```
("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 last primitives I can define with type `value list -> value` are the printing primitives.

S367d. ⟨*primitives for μScheme* :: S367b⟩+≡ (S372a) ◁ S367c S367e ▷

```
("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"
```

S367e. ⟨*primitives for μScheme* :: S367b⟩+≡ (S372a) ◁ S367d

```
("hash",  unaryOp (fn SYM s => NUM (fnvHash s)
                    | v => raise RuntimeError (valueString v ^
                                              " is not a symbol"))) ::
```

The `error` primitive is special because although it raises the `RuntimeError` exception, this behavior is expected, and therefore 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.

S367f. ⟨*utility functions for building primitives in μScheme* S367f⟩≡ (S372a)

```
                               errorPrimitive : exp * value list -> value list
fun errorPrimitive (_, [v]) = raise RuntimeError (valueString v)
  | errorPrimitive (e, vs)  = inExp (arityError 1) (e, vs)
```

O.2 Overall interpreter structure

O.2.1 A reusable read-eval-print loop

Functions `eval` and `evaldef` process expressions and true definitions. But an interpreter for μScheme also has to process the extended definitions `USE` and `TEST`, which need more tooling:

- To process a `USE`, we 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`), we must be able to run tests, and to run a test, we must call `eval`.

A lot of 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:

  S368a. ⟨*type* `interactivity` *plus related functions and value* S368a⟩≡ (S237a) S368b ▷
  ```
  datatype input_interactivity = PROMPTING | NOT_PROMPTING
  ```

  On output, an interpreter may or may not show a response to each definition.

  S368b. ⟨*type* `interactivity` *plus related functions and value* S368a⟩+≡ (S237a) ◁ S368a S368c ▷
  ```
  datatype output_interactivity = PRINTING | NOT_PRINTING
  ```

  Both kinds of information go to `processDef`, as a value of type `interactivity`.

S368c. ⟨*type* `interactivity` *plus related functions and value* S368a⟩+≡ (S237a) ◁ S368b
```
type interactivity =
  input_interactivity * output_interactivity
val noninteractive =
  (NOT_PROMPTING, NOT_PRINTING)
fun prompts (PROMPTING,     _) = true
  | prompts (NOT_PROMPTING, _) = false
fun prints (_, PRINTING)     = true
  | prints (_, NOT_PRINTING) = false
```

```
type interactivity
noninteractive : interactivity
prompts : interactivity -> bool
prints  : interactivity -> bool
```

When reading definitions of predefined functions, there's no interactivity.

S369a. ⟨*shared read-eval-print loop and* `processPredefined` S369a⟩≡ (S369b) S369c ▷

```
noninteractive    : interactivity
processPredefined : def * basis -> basis
```

```
fun processPredefined (def,basis) =
  processDef (def, basis, noninteractive)
```

- 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 S377.

If have these pieces, I can define one version of `processTests` (Section I.3 on page S247) and one read-eval-print loop, each of which is shared among many bridge languages. The pieces are organized as follows:

S369b. ⟨*evaluation, testing, and the read-eval-print loop for μScheme* S369b⟩≡ (S373a)

```
type basis
processDef   : def * basis * interactivity -> basis
testIsGood   : unit_test      * basis -> bool
processTests : unit_test list * basis -> unit
```

⟨*definitions of* `eval`, `evaldef`, `basis`, *and* `processDef` *for μScheme* S370c⟩
⟨*shared unit-testing utilities* S246d⟩
⟨*shared definition of* `withHandlers` S371a⟩
⟨*definition of* `testIsGood` *for μScheme* S378a⟩
⟨*shared definition of* `processTests` S247b⟩
⟨*shared read-eval-print loop and* `processPredefined` S369a⟩

Given `processDef` and `testIsGood`, function `readEvalPrintWith` processes a *stream* of extended definitions. A stream is like a list, except that when client code first looks at an element of a stream, the stream abstraction may do some input or output. 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 S238). 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.)

S369c. ⟨*shared read-eval-print loop and* `processPredefined` S369a⟩+≡ (S369b) ◁ S369a

```
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 S370b⟩
      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 S305e): 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 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:

- 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, we call function useFile, which reads definitions from a file and recursively passes them to readEvalPrintWith.

S370a. ⟨*definition of* useFile*, to read from a file* S370a⟩≡ (S370b)

```
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 on page 53. Function try passes each true definition DEF to function processDef, which does the language-dependent work.

S370b. ⟨*definition of* processXDef*, which can modify* unitTests *and call* errmsg S370b⟩≡ (S369c)

```
errmsg     : string -> unit
processDef : def * basis * interactivity -> basis
```

```
fun processXDef (xd, basis) =
  let ⟨definition of useFile, to read from a file S370a⟩
      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 bad definition, processXDef must recover from errors. 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. Function processDef calls evaldef, prints its response, and returns its environment.

S370c. ⟨*definitions of* eval*,* evaldef*,* basis*, and* processDef *for µScheme* S370c⟩≡ (S369b)

```
type basis = value ref env
fun processDef (d, rho, interactivity) =
  let val (rho', response) = evaldef (d, rho)
      val _ = if prints interactivity then println response else ()
  in  rho'
  end
```

A last word about `readEvalPrintWith`: you might be wondering, "where does it read, evaluate, and print?" Well, `readEvalPrintWith` doesn't do those things itself—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, what you do is call `readEvalPrintWith eprintln`, passing your extended definitions and your environments.

O.2.2 *Recovering from exceptions*

In normal execution, calling `withHandlers f a caught` applies function `f` to argument `a` and returns the result. But when the application of `f` raises an exception, `withHandlers` uses Standard ML's `handle` construct to recover from the exception and to pass an error message to `caught`, which acts as a failure continuation, as described in Section 2.10 on page 138. 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 `RuntimeError`, `NotFound`, and `Located`. Exceptions `RuntimeError` and `NotFound` are defined above; they signal problems with evaluation or with an environment, respectively. Exception `Located`, which is defined in Appendix I, is a special exception that wraps *another* exception `exn` in a source-code location. When `Located` is caught, we "re-raise" exception `exn`, and we fill in the source location in `exn`'s error message.

S371a. ⟨*shared definition of* `withHandlers` S371a⟩≡ (S369b)

`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 S371b⟩
```

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 while evaluating an expression or when reading a file.

S371b. ⟨*other handlers that catch non-fatal exceptions and pass messages to* `caught` S371b⟩≡ (S371a)

```
| 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)
```

I reuse the same exception handlers in later interpreters.

O.2.3 *Initializing and running the interpreter*

To get a complete interpreter running, what's left to do is what's done in C function `main` (page S309): decide if the interpreter is interactive, initialize the environment and the error format, and start the read-eval-print loop on the standard input. First, the initial environment.

A basis for µScheme comprises a single value environment. I create the initial basis by starting with the empty environment, binding the primitive operators, then reading the predefined functions. When reading predefined functions, the interpreter echoes no responses, and to issue error messages, it uses the special function `predefinedError`.

S372a. ⟨*implementations of µScheme primitives and definition of* `initialBasis` S372a⟩≡ (S373a)

```
⟨utility functions for building primitives in µScheme S367f⟩      initialBasis : basis
val initialBasis =
  let val rho =
        foldl (fn ((name, prim), rho) => bind (name, ref (PRIMITIVE (inExp prim)), rho))
              emptyEnv (⟨primitives for µScheme :: S367b⟩ [])
      val rho = bind ("error", ref (PRIMITIVE errorPrimitive), rho)
      val fundefs = ⟨predefined µScheme functions, as strings (from ⟨additions to the µScheme initial basis 98a⟩)⟩
      val xdefs = stringsxdefs ("predefined functions", fundefs)
  in  readEvalPrintWith predefinedFunctionError (xdefs, rho, noninteractive)
  end
```

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 `errorAt` and `synerrormsg` in Section I.5 on pages S254 and S256).

S372b. ⟨*shared utility functions for initializing interpreters* S372b⟩≡ (S237a)

```
fun setup_error_format interactivity =
  if prompts interactivity then
    toplevel_error_format := WITHOUT_LOCATIONS
  else
    toplevel_error_format := WITH_LOCATIONS
```

Function `runAs` looks at the interactivity mode and sets both the error format and the prompts. It then starts the read-eval-print loop on standard input, with the initial basis.

S372c. ⟨*function* `runAs`*, which evaluates standard input given* `initialBasis` S372c⟩≡ (S373a)

```
fun runAs interactivity =                          runAs : interactivity -> unit
  let val _ = setup_error_format interactivity
      val prompts = if prompts interactivity then stdPrompts else noPrompts
      val xdefs = filexdefs ("standard input", TextIO.stdIn, prompts)
  in  ignore (readEvalPrintWith eprintln (xdefs, initialBasis, interactivity))
  end
```

To launch the interpreter, I look at command-line arguments and call `runAs`. The code is executed only for its side effect, so I put it on the right-hand side of a `val` binding with no name. Function `CommandLine.arguments` returns an argument list; `CommandLine.name` returns the name by which the interpreter was invoked.

S372d. ⟨*code that looks at command-line arguments and calls* `runAs` *to run the interpreter* S372d⟩≡ (S373a)

```
val _ = case CommandLine.arguments ()
          of []     => runAs (PROMPTING,     PRINTING)
           | ["-q"] => runAs (NOT_PROMPTING, PRINTING)
           | _      => eprintln ("Usage: " ^ CommandLine.name () ^ " [-q]")
```

O.2.4 Pulling the pieces together in the right order

As mentioned in the introduction to this chapter, the ML language requires that every type and function be defined before it is used. Definitions come not only from this chapter but also from Appendices J and O. To get all the definitions in the right order, I use Noweb code chunks. The interpreters differ in detail, but each is put together 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. As shown in the next chapter, interpreters for typed languages also have chunks devoted to types and type checking (or type inference).

S373a. ⟨*mlscheme.sml* S373a⟩≡
  ⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S237a⟩

  ⟨*abstract syntax and values for μScheme* S365c⟩
  ⟨*utility functions on μScheme, Typed μScheme, and nano-ML values* S365d⟩

  ⟨*lexical analysis and parsing for μScheme, providing* `filexdefs` *and* `stringsxdefs` S373b⟩

  ⟨*evaluation, testing, and the read-eval-print loop for μScheme* S369b⟩

  ⟨*implementations of μScheme primitives and definition of* `initialBasis` S372a⟩
  ⟨*function* `runAs`*, which evaluates standard input given* `initialBasis` S372c⟩
  ⟨*code that looks at command-line arguments and calls* `runAs` *to run the interpreter* S372d⟩

O.3 Lexical analysis and parsing

Lexical analysis and parsing is implemented by these code chunks:

S373b. ⟨*lexical analysis and parsing for μScheme, providing* `filexdefs` *and* `stringsxdefs` S373b⟩≡
  ⟨*lexical analysis for μScheme and related languages* S373c⟩
  ⟨*parsers for single μScheme tokens* S374d⟩
  ⟨*parsers and parser builders for formal parameters and bindings* S375a⟩
  ⟨*parsers and parser builders for Scheme-like syntax* S375d⟩
  ⟨*parsers and* `xdef` *streams for μScheme* S376b⟩
  ⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S254c⟩

O.3.1 Tokens of the μScheme language

Our general parsing mechanism from Appendix J requires a language-specific `token` type and two functions `tokenString` and `isLiteral`.

S373c. ⟨*lexical analysis for μScheme and related languages* S373c⟩≡ (S373b) S373d ▷

```
datatype pretoken = QUOTE
                  | INT     of int
                  | SHARP   of bool
                  | NAME    of string
type token = pretoken plus_brackets
```

I define `isLiteral` by comparing the given string `s` with the string form of token `t`.

S373d. ⟨*lexical analysis for μScheme and related languages* S373c⟩+≡ (S373b) ◁ S373c S374a ▷

```
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.3.2 Lexical analysis for μScheme

Before a μScheme token, whitespace is ignored. 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.

S374a. ⟨*lexical analysis for μScheme and related languages* S373c⟩+≡ (S373b) ◁ S373d

```
schemeToken : token lexer
atom : string -> pretoken
```

```
local
  ⟨functions used in all lexers S374c⟩
  ⟨functions used in the lexer for μScheme S374b⟩
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.

S374b. ⟨*functions used in the lexer for μScheme* S374b⟩≡ (S374a)

```
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, we're 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 we encounter any other character, something has gone wrong. (The polymorphic type of `noneIfLineEnds` provides a subtle but powerful hint that no token can be produced; the only possible outcomes are that nothing is produced, or the lexer detects an error.)

S374c. ⟨*functions used in all lexers* S374c⟩≡ (S374a)

```
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.3.3 Parsers for μScheme

A parser consumes a stream of tokens and produces an abstract-syntax tree. The easiest way to write a parser is to begin with code for parsing the smallest things and finish with the code for parsing the biggest things. I parse tokens, literal S-expressions, μScheme expressions, and finally μScheme definitions.

Parsers for μScheme expressions

Usually a parser knows what kind of token it is looking for. To make such a parser easier to write, I create 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.

S374d. ⟨*parsers for single μScheme tokens* S374d⟩≡ (S373b)

```
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 name     = (fn (NAME  n)  => SOME n  | _ => NONE) <$>? pretoken
val any_name = name
```

The next step up is syntactic elements used in multiple Scheme-like languages. Function `formals` parses a list of formal parameters. If the formal parameters contain duplicates, it's treated as a syntax error. Function `bindings` produces a list of bindings suitable for use in `let*` expressions. For `let` and `letrec` expressions, which do not permit multiple bindings to the same name, use `distinctBsIn`.

S375a. ⟨*parsers and parser builders for formal parameters and bindings* S375a⟩≡ (S373b) S375b ▷

```
formalsOf   : string -> name parser -> string -> name list parser
bindingsOf  : string -> 'x parser -> 'e parser -> ('x * 'e) list parser
distinctBsIn : (name * 'e) list parser -> string -> (name * 'e) list parse
```

```
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

fun distinctBsIn bindings context =
  let fun check (loc, bs) =
        nodups ("bound name", context) (loc, map fst bs) >>=+ (fn _ => bs)
  in  check <$>! @@ bindings
  end
```

Record fields also may not contain duplicates.

S375b. ⟨*parsers and parser builders for formal parameters and bindings* S375a⟩+≡ (S373b) ◁ S375a S

```
recordFieldsOf : name parser -> name list parser
```

```
fun recordFieldsOf name =
  nodups ("record fields", "record definition") <$>!
                                    @@ (bracket ("(field ...)", many name))
```

We parse any keyword as the name represented by the same string as the keyword. And using the keyword parser, we can string together "usage" parsers.

S375c. ⟨*parsers and parser builders for formal parameters and bindings* S375a⟩+≡ (S373b) ◁ S375b

```
kw : string -> string parser
usageParsers : (string * 'a parser) list -> 'a parser
```

```
fun kw keyword =
  eqx keyword any_name
fun usageParsers ps = anyParser (map (usageParser kw) ps)
```

I'm now ready to parse a quoted S-expression, which is a symbol, a number, a Boolean, a list of S-expressions, or a quoted S-expression.

S375d. ⟨*parsers and parser builders for Scheme-like syntax* S375d⟩≡ (S373b) S376a ▷

```
sexp : value parser
```

```
fun sexp tokens = (
     SYM        <$> (notDot <$>! @@ any_name)
 <|> 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, ".") =
      errorAt "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 support this notation.

S376a. ⟨*parsers and parser builders for Scheme-like syntax* S375d⟩+≡ (S373b) ◁S375d S376d▷

```
fun atomicSchemeExpOf name =  VAR                         <$> name
                          <|> LITERAL <$> NUM         <$> int
                          <|> LITERAL <$> embedBool <$> booltok
```


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 J.3.4 on page S277: each alternative is specified by a pair containing a usage string and a parser.

S376b. ⟨*parsers and* `xdef` *streams for µScheme* S376b⟩≡ (S373b) S376e▷

```
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
  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  <$> dbs "letrec" <*> exp)
     , ("(let* (bindings) body)",   curry3 LETX LETSTAR <$> bindings     <*> exp)
     , ("(quote sexp)",             LITERAL             <$> sexp)
     ⟨rows added to ML µScheme's exptable in exercises S376c⟩
     ]
  end
```

There is a placeholder for adding more syntax in exercises.

S376c. ⟨*rows added to ML µScheme's* `exptable` *in exercises* S376c⟩≡ (S376b)

```
(* add syntactic sugar here, each row preceded by a comma *)
```

The `exp` 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.

S376d. ⟨*parsers and parser builders for Scheme-like syntax* S375d⟩+≡ (S373b) ◁S376a

```
fun fullSchemeExpOf atomic keywordsOf =
  let val exp = fn tokens => fullSchemeExpOf atomic keywordsOf tokens
  in      atomic
      <|> keywordsOf 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
```

S376e. ⟨*parsers and* `xdef` *streams for µScheme* S376b⟩+≡ (S373b) ◁S376b S377a▷

```
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. Function `define` is a Curried function that creates a `DEFINE` node.

S377a. ⟨*parsers and* `xdef` *streams for μScheme* S376b⟩+≡ (S373b) ◁S376e S377b▷

```
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 a unit test.

S377b. ⟨*parsers and* `xdef` *streams for μScheme* S376b⟩+≡ (S373b) ◁S377a S377c▷

```
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 on page 169.

S377c. ⟨*parsers and* `xdef` *streams for μScheme* S376b⟩+≡ (S373b) ◁S377b S377e▷

```
xdeftable : xdef parser
```

```
val xdeftable = usageParsers
  [ ("(use filename)", USE <$> name)
  ⟨rows added to μScheme xdeftable in exercises S377d⟩
  ]
```

S377d. ⟨*rows added to μScheme* `xdeftable` *in exercises* S377d⟩≡ (S377c)

```
(* add syntactic sugar here, each row preceded by a comma *)
```

The `xdef` parser combines all the types of extended definition, plus an error case.

S377e. ⟨*parsers and* `xdef` *streams for μScheme* S376b⟩+≡ (S373b) ◁S377c S377f▷

```
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.

S377f. ⟨*parsers and* `xdef` *streams for μScheme* S376b⟩+≡ (S373b) ◁S377e

```
xdefstream : string * line stream * prompts -> xdef stream
```

```
val xdefstream =
  interactiveParsedStream (schemeToken, xdef)
```

O.4 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 implementation is simple enough, and it uses enough named auxiliary functions—like `passes`, `checkExpectPasses` checkAssertPasses, and checkErrorPasses—that I always know from context what a Boolean value is supposed to mean. You might enjoy comparing the code below with the C code on pages S295 to S297, 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 S295.

In µScheme, a test is good if it passes. (In some other languages, tests must also be well typed.)

S378a. ⟨*definition of* testIsGood *for µScheme* S378a⟩≡ (S369b)

```
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 S378b⟩
      ⟨shared check{Expect,Assert,Error{Passes, which call outcome S246c⟩
      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, the only expressions that are syntactic values are literal expressions.

S378b. ⟨asSyntacticValue *for µScheme, Typed Impcore, Typed µScheme, and nano-ML* S378b⟩≡ (S378a)

```
asSyntacticValue : exp -> value option
```

```
fun asSyntacticValue (LITERAL v) = SOME v
  | asSyntacticValue _           = NONE
```

To print information about a failed test, we need function expString.

S378c. ⟨*definition of* expString *for µScheme* S378c⟩≡ (S365c)

```
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 e]
         | LAMBDA (xs, body) => bracketSpace ["lambda", bracketSpace xs, expString body]
  end
```

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 S318c, the ML version just cycles through a few different values. It's enough to prevent careless people from assuming that such a value is always `NIL`.

S379. ⟨*utility functions on μScheme, Typed μScheme, and nano-ML values* S365d⟩+≡ (S373a) ◁ S366b

```
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 _ => let exception Unspecified in raise Unspecifie
```

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 S374c.

Chapter contents

Supporting code for Typed Impcore

P

P.1 Predefined functions

As in Chapter 1, we define modulus in terms of division.

S381a. ⟨*predefined Typed Impcore functions* S381a⟩≡
```
(define int mod ([m : int] [n : int]) (- m (* n (/ m n))))
(define int negated ([n : int]) (- 0 n))
```

P.2 Unworthy interpreter code

The full story about abstract syntax: the definition of `xdef` is shared with μScheme, and functions `valueString` and `expString` are defined below.

S381b. ⟨*abstract syntax and values for Typed Impcore* S381b⟩≡ (S383a)
⟨*definitions of* `exp` *and* `value` *for Typed Impcore* 340f⟩
⟨*definition of type* `func`*, to represent a Typed Impcore function* 341e⟩
⟨*definition of* `def` *for Typed Impcore* 341c⟩
⟨*definition of* `unit_test` *for Typed Impcore* 341d⟩
⟨*definition of* `xdef` *(shared)* S365b⟩
⟨*definition of* `valueString` *for Typed Impcore* S386b⟩
⟨*definition of* `expString` *for Typed Impcore* S385b⟩
⟨*definitions of* `defString` *and* `defName` *for Typed Impcore* S385c⟩
⟨*definitions of functions* `toArray` *and* `toInt` *for Typed Impcore* 354a⟩

S381c. ⟨*definition of* `badParameter` S381c⟩≡ (350b)
```
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))
```

P.2.1 Processing definitions: typing and evaluation

Now that we can both type and evaluate definitions, we can define the type `topenv` and function `processDef` needed for Typed Impcore to work with the reusable read-eval-print loop described in Section O.2.1 on page S368. 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.

S382a. ⟨*definitions of* `basis` *and* `processDef` *for Typed Impcore* S382a⟩≡ (S388c)

```
                                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 prints interactivity then println (valstring ^ " : " ^ tystring)
              else ()
  in  (tglobals, tfuns, vglobals, vfuns)
  end
```

The distinction between "compile time," where we run the typing phase `typdef`, and "run time," where we run 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.

The definition of the evaluation function `evaldef` appears in Appendix P.

P.2.2 *The read-eval-print loop*

Typed Impcore reuses the read-eval-print loop defined in Section O.2.1 on page S368. But Typed Impcore needs handlers for new exceptions: `TypeError` and `BugInTypeChecking`. `TypeError` is raised not at parsing time, and not at evaluation time, but at type-checking time. `BugInTypeChecking` should never be raised.

S382b. ⟨*other handlers that catch non-fatal exceptions and pass messages to* `caught` S382b⟩≡

```
  | TypeError          msg => caught ("type error <at loc>: " ^ msg)
  | BugInTypeChecking msg => caught ("bug in type checking: " ^ msg)
```

S382c. ⟨*more handlers for* `atLoc` S382c⟩≡

```
  | e as TypeError _          => raise Located (loc, e)
  | e as BugInTypeChecking _  => raise Located (loc, e)
```

P.2.3 *Building the initial basis*

The initial basis includes both primitive and predefined functions.

S382d. ⟨*implementations of Typed Impcore primitives and definition of* `initialBasis` S382d⟩≡ (S383a)

```
  ⟨shared utility functions for building primitives in languages with type checking S389d⟩
  ⟨utility functions and types for making Typed Impcore primitives S389f⟩
  val initialBasis =
    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 :: S390a⟩ nil)
        val primBasis = (emptyEnv, tfuns, emptyEnv, vfuns)
        val fundefs   = ⟨predefined Typed Impcore functions, as strings (from chunk 340a)⟩
        val xdefs     = stringsxdefs ("predefined functions", fundefs)
    in  readEvalPrintWith predefinedFunctionError (xdefs, primBasis, noninteractive)
    end
```

The code for the primitives appears in Appendix P. It resembles the code in Chapter 5, but it supplies a type, not just a value, for each primitive.

P.2.4 *Pulling the pieces together*

The parts of the ML code are put together in much the same way as the parts of the interpreter for *μ*Scheme in ⟨*mlscheme.sml* S373a⟩. And there are two new chunks that have no counterpart in an interpreter for *μ*Scheme: ⟨*types for Typed Impcore* 340c⟩ and ⟨*type checking for Typed Impcore* 347a⟩.

S383a. ⟨*timpcore.sml* S383a⟩≡

⟨*exceptions used in languages with type checking* S237b⟩
⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S237a⟩

⟨*types for Typed Impcore* 340c⟩

⟨*abstract syntax and values for Typed Impcore* S381b⟩
⟨*utility functions on Typed Impcore values* S383b⟩

⟨*type checking for Typed Impcore* 347a⟩

⟨*lexical analysis and parsing for Typed Impcore, providing* `filexdefs` *and* `stringsxdefs` S386c⟩

⟨*evaluation, testing, and the read-eval-print loop for Typed Impcore* S388c⟩

⟨*implementations of Typed Impcore primitives and definition of* `initialBasis` S382d⟩
⟨*function* `runAs`, *which evaluates standard input given* `initialBasis` S372c⟩
⟨*code that looks at command-line arguments and calls* `runAs` *to run the interpreter* S372d⟩

P.3 Unit testing

S383b. ⟨*utility functions on Typed Impcore values* S383b⟩≡ (S383a)

```
fun testEqual (NUM n,    NUM n')   = n = n'
  | testEqual (ARRAY a, ARRAY a') = a = a'
  | testEqual (_,        _)        = false
```

S383c. ⟨*definition of* `testIsGood` *for Typed Impcore* S383c⟩≡ (S388c)

```
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 de
      fun deftystring d =
        let val (_, _, t) = typdef (d, tglobals, tfuns)
        in  t
        end handle NotFound x => raise TypeError ("name " ^ x ^ " is not definec
      ⟨shared check{Expect,Assert,Error,Type{Checks, which call ty S384d⟩
      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))) () (ERR
      ⟨asSyntacticValue for μScheme, Typed Impcore, Typed μScheme, and nano-ML S378b⟩
      ⟨shared check{Expect,Assert,Error{Passes, which call outcome S246c⟩
      ⟨shared checkTypePasses and checkTypeErrorPasses, which call ty S384b⟩
      ⟨definition of checkFunctionTypePasses S384a⟩
      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
```

S384a. ⟨*definition of* `checkFunctionTypePasses` S384a⟩≡ (S383c)

```
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]
```


S384b. ⟨*shared* `checkTypePasses` *and* `checkTypeErrorPasses`, *which call* `ty` S384b⟩≡ (S383c S401e) S384c ▷

```
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]
```

S384c. ⟨*shared* `checkTypePasses` *and* `checkTypeErrorPasses`, *which call* `ty` S384b⟩+≡ (S383c S401e) ◁ S384b

```
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)
```

S384d. ⟨*shared* `check{Expect,Assert,Error,Type{Checks`, *which call* `ty` S384d⟩≡ (S383c S401e) S385a ▷

```
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]
```

S385a. ⟨*shared* check{Expect,Assert,Error,Type{Checks, *which call* ty S384d⟩+≡ (S383c S401e)

```
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"
```

S385b. ⟨*definition of* expString *for Typed Impcore* S385b⟩≡ (S381b)

```
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, expStri
         | 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
```

S385c. ⟨*definitions of* defString *and* defName *for Typed Impcore* S385c⟩≡ (S381b)

```
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.4 Printing types and values

This code prints types.

S385d. ⟨*definitions of* typeString *and* funtyString *for Typed Impcore* S385d⟩≡ S386a ▷

```
fun typeString BOOLTY         = "bool"
  | typeString INTTY          = "int"
  | typeString UNITTY         = "unit"
  | typeString (ARRAYTY tau) = "(array " ^ typeString tau ^ ")"
```

S386a. ⟨*definitions of* `typeString` *and* `funtyString` *for Typed Impcore* S385d⟩+≡ ◁S385d

```
fun funtyString (FUNTY (args, result)) =
  "(" ^ spaceSep (map typeString args) ^ " -> " ^ typeString result ^ ")"
```

It would be good to figure out how to use `separate` in this code.

S386b. ⟨*definition of* `valueString` *for Typed Impcore* S386b⟩≡ (S381b)

```
fun valueString (NUM n) = intString n
  | valueString (ARRAY a) =
      if Array.length a = 0 then
          "[]"
      else
          let val elts = Array.foldr (fn (v, s) => " " :: valueString v :: s) ["]"] a
          in  String.concat ("[" :: tl elts)
          end
```

P.5 Parsing

Typed Impcore can use μScheme's lexical analysis, so all we have here is a parser.

S386c. ⟨*lexical analysis and parsing for Typed Impcore, providing* `filexdefs` *and* `stringsxdefs` S386c⟩≡ (S383a)

⟨*lexical analysis for μScheme and related languages* S373c⟩
⟨*parsers for single μScheme tokens* S374d⟩
⟨*parsers and parser builders for formal parameters and bindings* S375a⟩
⟨*parser builders for typed languages* S387a⟩
⟨*parsers and* `xdef` *streams for Typed Impcore* S386d⟩
⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S254c⟩

S386d. ⟨*parsers and* `xdef` *streams for Typed Impcore* S386d⟩≡ (S386c) S387b▷

```
exp      : exp parser
exptable : exp parser -> exp parser
```

```
val name    = sat (fn n => n <> "->") name  (* an arrow is not a name *)
val arrow   = (fn (NAME "->") => SOME () | _ => NONE) <$>? pretoken

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)
  ]

fun impcorefun what exp =  name
                       <|> exp <!> ("only named functions can be " ^ what)
                       <?> "function name"

val atomicExp =  VAR     <$> name
             <|> LITERAL <$> NUM <$> int
             <|> booltok <!> "Typed Impcore has no Boolean literals"
             <|> quote   <!> "Typed Impcore has no quoted literals"


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
```

S387a. ⟨*parser builders for typed languages* S387a⟩≡ (S386c S395a)

```
typedFormalsOf : string parser -> 'b parser -> 'a parser -> string -> (stri
```

```
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
```

S387b. ⟨*parsers and* `xdef` *streams for Typed Impcore* S386d⟩+≡ (S386c) ◁S386d S387c▷

```
fun repeatable_ty tokens = (
      BOOLTY <$ kw "bool"
  <|> UNITTY <$ kw "unit"
  <|> INTTY  <$ kw "int"
  <|> (fn (loc, n) => errorAt ("Cannot recognize name " ^ n ^ " as a type") loc)
      <$>! @@ name
  <|> usageParsers [("(array ty)", ARRAYTY <$> 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)
```

S387c. ⟨*parsers and* `xdef` *streams for Typed Impcore* S386d⟩+≡ (S386c) ◁S387b S387d▷

```
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 a unit test.

S387d. ⟨*parsers and* `xdef` *streams for Typed Impcore* S386d⟩+≡ (S386c) ◁S387c S387e▷

```
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 <$> ex
  , ("(check-function-type f (tau ... -> tau))",
                 curry CHECK_FUNCTION_TYPE <$> impcorefun "checked" exp <*> fun
  ]
```

S387e. ⟨*parsers and* `xdef` *streams for Typed Impcore* S386d⟩+≡ (S386c) ◁S387d S388a▷

```
val xdeftable = usageParsers
  [ ("(use filename)", USE <$> name)
  ⟨rows added to Typed Impcore xdeftable in exercises S388b⟩
  ]

val xdef =  DEF  <$> deftable
        <|> TEST <$> testtable
```

```
          <|>          xdeftable
          <|> badRight "unexpected right bracket"
          <|> DEF <$> EXP <$> exp
          <?> "definition"
```

S388a. ⟨*parsers and* `xdef` *streams for Typed Impcore* S386d⟩+≡ (S386c) ◁S387e

```
val xdefstream = interactiveParsedStream (schemeToken, xdef)
```

S388b. ⟨*rows added to Typed Impcore* `xdeftable` *in exercises* S388b⟩≡ (S387e)

```
(* add syntactic extensions here, each preceded by a comma *)
```


P.6 Evaluation

S388c. ⟨*evaluation, testing, and the read-eval-print loop for Typed Impcore* S388c⟩≡ (S383a)
⟨*definitions of* `eval` *and* `evaldef` *for Typed Impcore* S388d⟩
⟨*definitions of* `basis` *and* `processDef` *for Typed Impcore* S382a⟩
⟨*shared definition of* `withHandlers` S371a⟩
⟨*shared unit-testing utilities* S246d⟩
⟨*definition of* `testIsGood` *for Typed Impcore* S383c⟩
⟨*shared definition of* `processTests` S247b⟩
⟨*shared read-eval-print loop and* `processPredefined` S369a⟩

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.

S388d. ⟨*definitions of* `eval` *and* `evaldef` *for Typed Impcore* S388d⟩≡ (S388c) S388e▷

```
val unitVal = NUM 1983
```

`ev : exp -> value`

The implementation of the evaluator uses the same techniques we use to implement μScheme in Chapter 5. Because of Typed Impcore's many environments, the evaluator does more bookkeeping.

S388e. ⟨*definitions of* `eval` *and* `evaldef` *for Typed Impcore* S388d⟩+≡ (S388c) ◁S388d S389b▷

`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 S389a⟩)
      ⟨more alternatives for ev for Typed Impcore 354b⟩
  in  ev e
  end
```

To apply a function, we build an evaluation environment. We strip the types off the formals and we put the actuals in mutable ref 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, we catch exception `BindListLength` and raise `BugInTypeChecking`.

S389a. ⟨*apply user-defined function* `func` *to* `args` S389a⟩≡ (S388e)

```
formals : name      list
actuals : value ref list
```

```
let val (formals, body) = func
    val actuals         = map (ref o ev) args
in  eval (body, globals, functions, bindList (formals, actuals, emptyEnv))
    handle BindListLength =>
        raise BugInTypeChecking "Wrong number of arguments to function"
end
```

Evaluating a definition produces two environments, plus a string representing the thing defined.

S389b. ⟨*definitions of* `eval` *and* `evaldef` *for Typed Impcore* S388d⟩+≡ (S388c) ◁S388e

```
evaldef : def * value ref env * func env -> value ref env * func env * st
```

```
fun evaldef (d, globals, functions) =
  case d
    of VAL (x, e) => ⟨evaluate e and bind the result to x S389c⟩
     | 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)
```

S389c. ⟨*evaluate* `e` *and bind the result to* `x` S389c⟩≡ (S389b)

```
let val v = eval (e, globals, functions, emptyEnv)
in  (bind (x, ref v, globals), functions, valueString v)
end
```

Here are the primitives. As in Chapter 5, all are either binary or unary operators. Type checking should guarantee that operators are used with the correct arity.

S389d. ⟨*shared utility functions for building primitives in languages with type checking* S389d⟩≡ (S3

```
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.

S389e. ⟨*shared utility functions for building primitives in languages with type checking* S389d⟩+≡ (

```
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")
```

S389f. ⟨*utility functions and types for making Typed Impcore primitives* S389f⟩≡ (S382d) S390c ▷

```
arithtype : funty
```

```
val arithtype = FUNTY ([INTTY, INTTY], INTTY)
```

As in Chapter 5, we use the chunk ⟨*primitive functions for Typed Impcore* :: S390a⟩ to cons up all the primitives into one giant list, and we use that list to build the initial environment for the read-eval-print loop. The big difference is that in Typed Impcore, each primitive has a type as well as a value.

S390a. ⟨*primitive functions for Typed Impcore* :: S390a⟩≡ (S382d) S390b ▷

```
("+", arithOp op +,   arithtype) ::
("-", arithOp op -,   arithtype) ::
("*", arithOp op *,   arithtype) ::
("/", arithOp op div, arithtype) ::
```


And printing Unicode.

S390b. ⟨*primitive functions for Typed Impcore* :: S390a⟩+≡ (S382d) ◁ S390a S390d ▷

```
("printu", unaryOp (fn (NUM n) => (printUTF8 n; unitVal)
                     | _ => raise BugInTypeChecking "printu of non-number"),
              FUNTY ([INTTY], UNITTY)) ::
```

Comparisons take two arguments. Most comparisons (except for equality) apply only to integers.

S390c. ⟨*utility functions and types for making Typed Impcore primitives* S389f⟩+≡ (S382d) ◁ S389f

```
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)
```

S390d. ⟨*primitive functions for Typed Impcore* :: S390a⟩+≡ (S382d) ◁ S390b

```
("<", intcompare op <, comptype) ::
(">", intcompare op >, comptype) ::
```

arithOp	S389e
arithtype	S389f
binaryOp	S389d
BOOLTY	340c
BugInTypeChecking	S237b
FUNTY	340c
INTTY	340c
NUM	340f
printUTF8	S239b
unaryOp	S389d
UNITTY	340c
unitVal	S388d

Chapter contents

Supporting code for the Typed μScheme interpreter

Q

Q.1 Master interpreter fragments

Unit tests are as for Typed Impcore, except we can check the type of any expression, not just a function.

S393a. ⟨*definition of* `unit_test` *for explicitly typed languages* S393a⟩≡ (S393b)

```
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
```

These pieces are pulled together as follows. The definition of `xdef` is, as usual, shared, and less usually, the definition of `valueString` is shared with μScheme and nano-ML.

S393b. ⟨*abstract syntax and values for Typed μScheme* S393b⟩≡ (S394b)

⟨*definitions of* `exp` *and* `value` *for Typed μScheme* 370a⟩
⟨*definition of* `def` *for Typed μScheme* 370c⟩
⟨*definition of* `unit_test` *for explicitly typed languages* S393a⟩
⟨*definition of* `xdef` *(shared)* S365b⟩
⟨*definition of* `valueString` *for μScheme, Typed μScheme, and nano-ML* 314⟩
⟨*definition of* `expString` *for Typed μScheme* S402b⟩
⟨*definitions of* `defString` *and* `defName` *for Typed μScheme* S403⟩

Q.1.1 *Infinite stream of type variables*

Stream `infiniteTyvars` is built from stream `naturals`, which contains the natural numbers; `naturals` is defined in chunk S252a in Appendix I.

S393c. ⟨*infinite supply of type variables* S393c⟩≡ (379a)

```
naturals       : int stream
infiniteTyvars : name stream
```

```
val infiniteTyvars =
  streamMap (fn n => "'b-" ^ intString n) naturals
```

Processing definitions in two phases

S393d. ⟨*definitions of* `basis` *and* `processDef` *for Typed μScheme* S393d⟩≡ (S397e)

```
processDef : def * basis * interactivity -> basis
```

⟨*definition of* `basis` *for Typed μScheme* 391d⟩

```
fun processDef (d, (Delta, Gamma, rho), interactivity) =
  let val (Gamma, tystring)  = typdef (d, Delta, Gamma)
      val (rho,   valstring) = evaldef (d, rho)
      val _ = if prints interactivity then
                println (valstring ^ " : " ^ tystring)
              else
                ()
  in  (Delta, Gamma, rho)
```

```
  end
```

Building the initial basis and interpreter

S394a. ⟨*implementations of Typed μScheme primitives and definition of* `initialBasis` S394a⟩≡ (S394b)

```
⟨shared utility functions for building primitives in languages with type checking S389d⟩
⟨utility functions and types for making Typed μScheme primitives S400a⟩
⟨definition of primBasis for Typed μScheme 391e⟩
val initialBasis =
  let val fundefs   = ⟨predefined Typed μScheme functions, as strings (from chunk S400e)⟩
      val xdefs     = stringsxdefs ("predefined functions", fundefs)
  in  readEvalPrintWith predefinedFunctionError (xdefs, primBasis, noninteractive)
  end
```


The primitives appear in Section Q.5 on page S399. They resemble the primitives in Chapter 5, except that each primitive comes with a type as well as a value.

Pulling the pieces together

The overall structure of the Typed μScheme interpreter is similar to the structure of the Typed Impcore interpreter, with the addition of kinds and kind checking.

S394b. ⟨*tuscheme.sml* S394b⟩≡

```
⟨exceptions used in languages with type checking S237b⟩
⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization S237a⟩

⟨kinds for typed languages 364a⟩
⟨types for Typed μScheme S394c⟩
⟨sets of free type variables in Typed μScheme 381a⟩
⟨shared utility functions on sets of type variables generated automatically⟩
⟨kind checking for Typed μScheme 387b⟩

⟨abstract syntax and values for Typed μScheme S393b⟩
⟨utility functions on μScheme, Typed μScheme, and nano-ML values 315a⟩

⟨capture-avoiding substitution for Typed μScheme 384a⟩
⟨type equivalence for Typed μScheme 379a⟩
⟨type checking for Typed μScheme generated automatically⟩

⟨lexical analysis and parsing for Typed μScheme, providing filexdefs and stringsxdefs S395a⟩

⟨evaluation, testing, and the read-eval-print loop for Typed μScheme S397e⟩

⟨implementations of Typed μScheme primitives and definition of initialBasis S394a⟩
⟨function runAs, which evaluates standard input given initialBasis S372c⟩
⟨code that looks at command-line arguments and calls runAs to run the interpreter S372d⟩
```

Q.2 Printing types and values

This code prints types. It might be desirable to print them using a more ML-like syntax.

S394c. ⟨*types for Typed μScheme* S394c⟩≡ (S394b)

```
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 ^ ")"
```

Q.3 Parsing

S395a. ⟨*lexical analysis and parsing for Typed µScheme, providing* `filexdefs` *and* `stringsxdefs` S395a⟩≡ (S394b)
⟨*lexical analysis for µScheme and related languages* S373c⟩
⟨*parsers for single µScheme tokens* S374d⟩
⟨*parsers for Typed µScheme tokens* S395b⟩
⟨*parsers and parser builders for formal parameters and bindings* S375a⟩
⟨*parsers and parser builders for Scheme-like syntax* S375d⟩
⟨*parser builders for typed languages* S395e⟩
⟨*parsers and* `xdef` *streams for Typed µScheme* S395c⟩
⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S254c⟩

S395b. ⟨*parsers for Typed µScheme tokens* S395b⟩≡ (S395a) S395d ▷

```
val arrow   = (fn (NAME "->") => SOME () | _ => NONE) <$>? pretoken
val name    = sat (fn n => n <> "->") name  (* an arrow is not a name *)
```

S395c. ⟨*parsers and* `xdef` *streams for Typed µScheme* S395c⟩≡ (S395a) S396a ▷

```
fun keyword words =
  let fun isKeyword s = List.exists (fn s' => s = s') words
  in  sat isKeyword name
  end

val expKeyword = keyword ["if", "while", "set", "begin", "lambda",
                          "type-lambda", "let", "let*", "@"]
val tyKeyword  = keyword ["forall", "function", "->"]

val tlformals = nodups ("formal type parameter", "type-lambda") <$>! @@ (many na

fun nodupsty what (loc, xts) = nodups what (loc, map fst xts) >>=+ (fn _ => xts)
                                                  (* error on duplicate na

fun letDups LETSTAR (_, bindings) = OK bindings
  | letDups LET     bindings      = nodupsty ("bound variable", "let") bindings
```

When parsing a type, we reject anything that looks like an expression.

S395d. ⟨*parsers for Typed µScheme tokens* S395b⟩+≡ (S395a) ◁ S395b

`tyvar : string parser`

```
val tyvar =
  quote *> (curry op ^ "'" <$> name <?> "type variable (got quote mark)")
```

S395e. ⟨*parser builders for typed languages* S395e⟩≡ (S395a S386c)

`distinctTyvars : name list parser`

```
val distinctTyvars =
  nodups ("quantified type variable", "forall") <$>! @@ (many tyvar)

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 functior
        | arrows args (_::_::_) = ERROR "multiple arrows in function type"
  in  fn xs => errorLabel "syntax error: " o arrows xs
  end
```

S396a. ⟨*parsers and* `xdef` *streams for Typed μScheme* S395c⟩+≡ (S395a) ◁S395c S396b▷

`ty : tyex parser`

```
val arrows = arrowsOf CONAPP FUNTY
fun ty tokens =
  let fun badExpKeyword (loc, bad) =
        errorAt ("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
```

When parsing an expression, we reject anything that looks like a type.

S396b. ⟨*parsers and* `xdef` *streams for Typed μScheme* S395c⟩+≡ (S395a) ◁S396a S397a▷

```
fun flipPair tau x = (x, tau)
val formal = bracket ("[x : ty]", pair <$> name <* kw ":" <*> ty)
val lformals = bracket ("([x : ty] ...)", many formal)
val tformals = bracket ("('a ...)", many tyvar)

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 name, 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 <$> tybindings <*> exp)
 <|> cb "let*"   "(let* (bindings) body)" (letx   LETSTAR <$> @@ 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 bindings ts = bindingsOf "(x e)" name exp ts
```

```
and badTyKeyword (loc, bad) =
      errorAt ("looking for expression but found `" ^ bad ^ "'") loc
```

The true-definition special forms.

S397a. ⟨*parsers and* `xdef` *streams for Typed μScheme* S395c⟩+≡ (S395a) ◁S396b S397b▷

```
fun define tau f formals body =
  nodupsty ("formal parameter", "definition of function " ^ f) formals >>=+ (fn
  DEFINE (f, tau, (xts, body)))

fun valrec (x, tau) e = VALREC (x, tau, e)

val def =
     cb "define" "(define type f (args) body)"
                                    (define <$> ty <*> name <*> @@ lformals <*
 <|> cb "val"     "(val x e)"              (curry VAL <$> name    <*> exp)
 <|> cb "val-rec" "(val-rec [x : type] e)"  (valrec    <$> formal <*> exp)
```

Function `unit_test` parses a unit test.

S397b. ⟨*parsers and* `xdef` *streams for Typed μScheme* S395c⟩+≡ (S395a) ◁S397a S397c▷

`unit_test : unit_test parser`

```
val unit_test =
      cb "check-expect" "(check-expect e1 e2)" (curry CHECK_EXPECT <$> exp <*> e
  <|> cb "check-assert" "(check-assert e)"     (      CHECK_ASSERT <$> exp)
  <|> cb "check-error"  "(check-error e)"      (      CHECK_ERROR  <$> exp)
  <|> cb "check-type"   "(check-type e tau)"   (curry CHECK_TYPE   <$> exp <*> t
  <|> cb "check-type-error" "(check-type-error e)"
                                     (CHECK_TYPE_ERROR <$> (def <|> EXP <$
```

And `xdef` parses extended definitions.

S397c. ⟨*parsers and* `xdef` *streams for Typed μScheme* S395c⟩+≡ (S395a) ◁S397b S397d▷

`xdef : xdef parser`

```
val xdef =
     DEF <$> def
 <|> cb "use" "(use filename)" (USE <$> name)
 <|> TEST <$> unit_test
 <|> badRight "unexpected right bracket"
 <|> DEF <$> EXP <$> exp
 <?> "definition"
```

S397d. ⟨*parsers and* `xdef` *streams for Typed μScheme* S395c⟩+≡ (S395a) ◁S397c

```
val xdefstream = interactiveParsedStream (schemeToken, xdef)
```

Q.4 Evaluation

S397e. ⟨*evaluation, testing, and the read-eval-print loop for Typed μScheme* S397e⟩≡ (S394b)

⟨*definition of* `namedValueString` *for functional bridge languages* S399c⟩
⟨*definitions of* `eval` *and* `evaldef` *for Typed μScheme* S398a⟩
⟨*definitions of* `basis` *and* `processDef` *for Typed μScheme* S393d⟩
⟨*shared definition of* `withHandlers` S371a⟩
⟨*shared unit-testing utilities* S246d⟩
⟨*definition of* `testIsGood` *for Typed μScheme* S401e⟩
⟨*shared definition of* `processTests` S247b⟩
⟨*shared read-eval-print loop and* `processPredefined` S369a⟩

The implementation of the evaluator is almost identical to the implementation in Chapter 5. There are only two significant differences: we have to deal with the mismatch in representations between the abstract syntax `LAMBDA` and the value `CLOSURE`, and we have to write cases for the `TYAPPLY` and `TYLAMBDA` expressions.

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, we raise the exception `BugInTypeChecking`, not `RuntimeError`.

S398a. ⟨*definitions of* `eval` *and* `evaldef` *for Typed μScheme* S398a⟩≡ (S397e) S399b ▷

```
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 389c⟩
        ⟨more alternatives for ev for Typed μScheme S398b⟩
  in  ev e
  end
```


Code for variables is just as in Chapter 5.

S398b. ⟨*more alternatives for* `ev` *for Typed μScheme* S398b⟩≡ (S398a) S398c ▷

```
| 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.

S398c. ⟨*more alternatives for* `ev` *for Typed μScheme* S398b⟩+≡ (S398a) ◁ S398b S398d ▷

```
| 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.

S398d. ⟨*more alternatives for* `ev` *for Typed μScheme* S398b⟩+≡ (S398a) ◁ S398c S398e ▷

```
| 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, we raise `BugInTypeChecking`, not `RuntimeError`, because the type checker should reject any program that could apply a non-function.

S398e. ⟨*more alternatives for* `ev` *for Typed μScheme* S398b⟩+≡ (S398a) ◁ S398d S398f ▷

```
| ev (APPLY (f, args))  =
      (case ev f
         of PRIMITIVE prim => prim (map ev args)
          | CLOSURE clo => ⟨apply closure clo to args 317b⟩
          | v => raise BugInTypeChecking "applied non-function"
      )
```

Code for the `LETX` family is as in Chapter 5.

S398f. ⟨*more alternatives for* `ev` *for Typed μScheme* S398b⟩+≡ (S398a) ◁ S398e S399a ▷

```
| 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 ((n, e), rho) = bind (n, ref (eval (e, rho)), rho)
    in  eval (body, foldl step rho bs)
    end
```

S399a. ⟨*more alternatives for* `ev` *for Typed μScheme* S398b⟩+≡ (S398a) ◁S398f

```
  | ev (LETRECX (bs, body)) =
      let val (tynames, values) = ListPair.unzip bs
          val names = map fst tynames
          val _ = errorIfDups ("bound name", names, "letrec")
          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 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.

S399b. ⟨*definitions of* `eval` *and* `evaldef` *for Typed μScheme* S398a⟩+≡ (S397e) ◁S398a

```
                                  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` is still bound to `NIL`—that is, before the assignment to `find (name, rho)`. Therefore, as described on page 371, evaluating `e` must not evaluate `name`—because the mutable cell for `name` does not yet contain its correct value.

The string returned by `evaldef` is the value, unless the value is a named procedure, in which case it is the name.

S399c. ⟨*definition of* `namedValueString` *for functional bridge languages* S399c⟩≡ (S397e)

```
  fun namedValueString x v =                namedValueString : name -> value -> string
    case v of CLOSURE _ => x
            | PRIMITIVE _ => x
            | _ => valueString v
```

Q.5 Primitives of Typed μScheme

Comparisons take two arguments. Most comparisons (but not equality) apply only to integers.

S400a. ⟨*utility functions and types for making Typed μScheme primitives* S400a⟩≡ (S394a)

```
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)
```

S400b. ⟨*primitive functions for Typed μScheme* :: S400b⟩≡ (391e) S400c ▷

```
("<", intcompare op <, comptype) ::
(">", intcompare op >, comptype) ::
("=", comparison equalatoms,
                       FORALL (["'a"], FUNTY ([tvA, tvA], booltype))) ::
```

Two of the print primitives also have polymorphic types.

S400c. ⟨*primitive functions for Typed μScheme* :: S400b⟩+≡ (391e) ◁ S400b

```
("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 μScheme, all the primitives are functions, so this chunk is empty. But you might add to it in the Exercises.

S400d. ⟨*primitives that aren't functions, for Typed μScheme* :: S400d⟩≡ (391e)

Q.6 Predefined functions

Because programming in Typed μScheme is an awful lot of trouble, Typed μScheme has fewer predefined functions than μScheme. Some of these functions are defined in Chapter 6. The rest are here.

Becauses lists in Typed μScheme are homogeneous, the funny list functions built from car and cdr are much less useful than in μScheme.

S400e. ⟨*predefined Typed μScheme functions* S400e⟩≡ S400f ▷

```
(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 almost exactly as in Typed Impcore.

S400f. ⟨*predefined Typed μScheme functions* S400e⟩+≡ ◁ S400e S401a ▷

```
(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))
```

Here is list append.

S401a. ⟨*predefined Typed μScheme functions* S400e⟩+≡ ◁S400f S401b▷

```
(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)))
```

In Typed μScheme, an association list must be represented as a list of pairs. The only sensible way to write a lookup function for an association list is to use continuation-passing style. These problems are given as exercises.

I provide just some of the list functions found in μScheme. Both `exists?` and `all?` are left as exercises. Function `foldr` is also given as an exercise.

Integer comparisons are as in Typed Impcore, but to define `!=` we need a type abstraction. This is progress! In Typed Impcore, a polymorphic `!=` can't be defined as as function.

S401b. ⟨*predefined Typed μScheme functions* S400e⟩+≡ ◁S401a S401c▷

```
(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.

S401c. ⟨*predefined Typed μScheme functions* S400e⟩+≡ ◁S401b

```
(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))))
```

Q.7 Unit testing

S401d. ⟨*utility functions on μScheme, Typed μScheme, and nano-ML values* **[[tuscheme]]** S401d⟩≡

```
fun testEqual (ARRAY a1, ARRAY a2) =
      Array.length a1 = Array.length a1 andalso
      Array.foldli (fn (i, v, equal) => equal andalso testEqual (v, Array.sub (a
  | testEqual (PAIR (car1, cdr1), PAIR (car2, cdr2)) =
      testEqual (car1, car2) andalso testEqual (cdr1, cdr2)
  | testEqual (v1, v2) = equalatoms (v1, v2)
```

S401e. ⟨*definition of* `testIsGood` *for Typed μScheme* S401e⟩≡ (S397e)

```
fun testIsGood (test, (Delta, Gamma, rho)) =
  let fun ty e = typeof (e, Delta, Gamma)
                 handle NotFound x => raise TypeError ("name " ^ x ^ " is not de
      ⟨shared check{Expect,Assert,Error,Type{Checks, which call ty S402a⟩
      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 str
      ⟨asSyntacticValue for μScheme, Typed Impcore, Typed μScheme, and nano-ML S378b⟩
```

```
  ⟨shared check{Expect,Assert,Error{Passes, which call outcome S246c⟩
  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 S384b⟩
  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
```

Testing forms `check-expect`, `check-error`, and `check-type` should contain only expressions that typecheck. But the whole point of `check-type-error` is that its expression *doesn't* typecheck. Thus, we don't typecheck it with the others—instead, like `check-type`, whether it has a type determines if it *passes*.

S402a. ⟨*shared* `check{Expect,Assert,Error,Type{Checks`, *which call* `ty` S402a⟩≡ (S401e S383c)

```
fun checkTypeChecks (e, tau) =
  let val tau' = ty e
  in  true
  end
  handle TypeError msg =>
         failtest ["In (check-type ", expString e, " " ^ typeString tau, "), ", msg]
```

S402b. ⟨*definition of* `expString` *for Typed μScheme* S402b⟩≡ (S393b)

```
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
```

S403. ⟨*definitions of* `defString` *and* `defName` *for Typed μScheme* S403⟩≡ (S393b)

```
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
fun defName (VAL (x, _))       = x
  | defName (VALREC (x, _, _)) = x
  | defName (DEFINE (x, _, _)) = x
  | defName (EXP _) = raise InternalError "asked for name defined by expression"
```

Chapter contents

Supporting code for nano-ML

R

R.1 Small pieces of the interpreter

S405a. ⟨*if* e *is not a* LAMBDA*, raise* TypeError S405a⟩≡ (450b)

```
case e of LAMBDA _ => ()
        | _ => raise TypeError ("in val-rec, right-hand side " ^ expString e ^
                                " is not a lambda")
```

Abstract syntax and values

Unit tests resemble the unit tests for Typed μScheme, but as explained in Section 7.4.6, the typing tests are subtly different. These unit tests are shared with other languages that use Hindley-Milner types.

S405b. ⟨*definition of* unit_test *for languages with Hindley-Milner types* S405b⟩≡ (S405c)

```
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
```

Here are all the pieces related to abstract syntax and values. As usual, `xdef` and `valueString` are shared with other languages. Function `expString` is defined in Appendix R.

S405c. ⟨*abstract syntax and values for nano-ML* S405c⟩≡ (S411c)

```
⟨definitions of exp and value for nano-ML 414⟩
⟨definition of def for nano-ML 415a⟩
⟨definition of unit_test for languages with Hindley-Milner types S405b⟩
⟨definition of xdef (shared) S365b⟩
⟨definition of valueString for μScheme, Typed μScheme, and nano-ML 314⟩
⟨definition of expString for nano-ML and μML S417a⟩
⟨definitions of defString and defName for nano-ML and μML S417c⟩
```

S405d. ⟨*utility functions on type constraints* S405d⟩≡ (S405e)

```
⟨definitions of constraintString and untriviate S412a⟩
```

`constraintString : con -> string`	
`untriviate       : con -> con`	

S405e. ⟨*type inference for nano-ML and μML* S405e⟩≡ (S411c)

```
⟨representation of type constraints 446a⟩
⟨utility functions on type constraints S405d⟩
⟨constraint solving 447d⟩
⟨exhaustiveness analysis for μML S419f⟩
⟨definitions of typeof and typdef for nano-ML and μML 448c⟩
```

`typeof  : exp * type_env -> ty * con`	
`typdef : def * type_env -> type_env * string`	

R.1.1 Evaluation

The gross structure of the evaluator for nano-ML is the same as the gross structure of the evaluator for μScheme. What's needed are the usual definitions of `eval`, `evaldef`, `basis`, and `processDef`. Language-specific testing code appears in Appendix R, and everything else is shared.

S406a. ⟨*evaluation, testing, and the read-eval-print loop for nano-ML* S406a⟩≡ (S411c)

```
⟨definition of namedValueString for functional bridge languages S399c⟩
⟨definitions of eval and evaldef for nano-ML and μML S406b⟩
⟨definitions of basis and processDef for nano-ML S410b⟩
⟨shared definition of withHandlers S371a⟩
⟨shared unit-testing utilities S246d⟩
⟨definition of testIsGood for nano-ML S414c⟩
⟨shared definition of processTests S247b⟩
⟨shared read-eval-print loop and processPredefined S369a⟩
```

Evaluation of expressions

Because the abstract syntax of nano-ML is a subset of μScheme, the evaluator is almost a subset of the μScheme evaluator. One difference is that because nano-ML doesn't have mutation, environments map names to values, instead of mapping them to mutable cells. Another is that type inference should eliminate most potential errors. If one of those errors occurs anyway, we raise the exception `BugInTypeInference`.

S406b. ⟨*definitions of* `eval` *and* `evaldef` *for nano-ML and μML* S406b⟩≡ (S406a) S407d ▷

`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, fn _ => 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 S406c⟩
               | _ => raise BugInTypeInference "Applied non-function"
               )
        ⟨more alternatives for ev for nano-ML and μML S407a⟩
  in  ev e
  end
```

To apply a closure, we bind formal parameters directly to the values of actual parameters, not to mutable cells.

S406c. ⟨*apply closure* `clo` *to* `args` S406c⟩≡ (S406b)

```
let val ((formals, body), mkRho) = clo
    val actuals = map ev args
in  eval (body, bindList (formals, actuals, mkRho ()))
    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.

S407a. ⟨*more alternatives for* `ev` *for nano-ML and μML* S407a⟩≡ (S406b) S407b ▷

```
  | ev (LETX (LET, bs, body)) =
      let val (names, values) = ListPair.unzip bs
      in  eval (body, bindList (names, map ev values, rho))
      end
```

LETSTAR evaluates pairs in sequence, adding a binding to ρ after each evaluation.

S407b. ⟨*more alternatives for* `ev` *for nano-ML and μML* S407a⟩+≡ (S406b) ◁ S407a S407c ▷

```
  | 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 right-hand sides must be lambda abstractions.

S407c. ⟨*more alternatives for* `ev` *for nano-ML and μML* S407a⟩+≡ (S406b) ◁ S407b

```
  | ev (LETX (LETREC, bs, body)) =
      let fun makeRho' () =
            let fun step ((x, e), rho) =
                      (case e
                         of LAMBDA l => bind (x, CLOSURE (l, makeRho'), rho)
                          | _ => raise BugInTypeInference "non-lambda in letrec")
            in  foldl step rho bs
            end
      in  eval (body, makeRho'())
      end
```

Evaluating definitions

Evaluating a definition can produce a new environment. Function `evaldef` also returns a string that gives the name or value being defined.

S407d. ⟨*definitions of* `eval` *and* `evaldef` *for nano-ML and μML* S406b⟩+≡ (S406a) ◁ S406b S408a ▷

```
fun evaldef (VAL (x, e), rho) =      evaldef : def * value env -> value env * string
      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 fun makeRho' () = bind (f, CLOSURE (lambda, makeRho'), rho)
          val v           = CLOSURE (lambda, makeRho')
      in  (makeRho'(), f)
      end
  | evaldef (VALREC _, rho) =
      raise BugInTypeInference "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 for which we can compute the value without having the environment.

As in the type system, `DEFINE` is syntactic sugar for a combination of `VALREC` and `LAMBDA`.

S408a. ⟨*definitions of* `eval` *and* `evaldef` *for nano-ML and μML* S406b⟩+≡ (S406a) ◁S407d

```
  | evaldef (DEFINE (f, lambda), rho) =
      evaldef (VALREC (f, LAMBDA lambda), rho)
```

⟨*clause for* `evaldef` *for datatype definition (μML only)* S408b⟩

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.

S408b. ⟨*clause for* `evaldef` *for datatype definition (μML only)* S408b⟩≡ (S408a)

```
(* code goes here in Chapter 11 *)
```

R.1.2 A complete infrastructure for Hindley-Milner types

The sections above make a foundation on which we can implement constraint solving and type inference. The pieces are pulled together here.

S408c. ⟨*Hindley-Milner types with named type constructors* S408c⟩≡ (S411c)

⟨*definitions of* `tycon`*,* `eqTycon`*, and* `tyconString` *for named type constructors* 419a⟩
⟨*representation of Hindley-Milner types* 418⟩
⟨*sets of free type variables in Hindley-Milner types* 442⟩

```
val funtycon = "function"
```

⟨*functions that create or compare Hindley-Milner types with named type constructors* 422c⟩
⟨*definition of* `typeString` *for Hindley-Milner types* S411d⟩
⟨*shared utility functions on Hindley-Milner types* S412b⟩
⟨*specialized environments for type schemes* 446a⟩

R.1.3 Primitives

Arithmetic primitives expect and return integers. 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 them all at one go when we create the initial environment.

S408d. ⟨*shared utility functions for building primitives in languages with type inference* S408d⟩≡ (S411b)

```
unaryOp  : (value          -> value) -> (value list -> value)
binaryOp : (value * value -> value) -> (value list -> value)
arithOp  : (int * int -> int) -> (value list -> value)
arithtype : ty
```

```
fun binaryOp f = (fn [a, b] => f (a, b) | _ => raise BugInTypeInference "arity 2")
fun unaryOp  f = (fn [a]    => f  a     | _ => raise BugInTypeInference "arity 1")
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)
```

Here are some arithmetic primitives:

S408e. ⟨*primitives for nano-ML and μML* `::` S408e⟩≡ (S411b) S409b ▷

```
("+", arithOp op +,   arithtype) ::
("-", arithOp op -,   arithtype) ::
("*", arithOp op *,   arithtype) ::
("/", arithOp op div, arithtype) ::
```

Nano-ML has two kinds of predicates: `null?` takes one argument, and comparisons take two. Some comparisons apply only to integers. The supporting functions reuse `embedBool`.

S409a. ⟨*utility functions for building nano-ML primitives* S409a⟩≡ (S411b)

```
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 $\alpha \times \alpha \to$ `bool`, which gets generalized to type scheme $\forall \alpha . \alpha \times \alpha \to$ `bool`. In full ML, values of function types may not be compared for equality.

S409b. ⟨*primitives for nano-ML and* μ*ML* `::` S408e⟩+≡ (S411b) ◁ S408e S409d ▷

```
("<", intcompare op <,             comptype inttype) ::
(">", intcompare op >,             comptype inttype) ::
("=", comparison primitiveEquality, comptype alpha) ::
```

S409c. ⟨*utility functions on* μ*Scheme, Typed* μ*Scheme, and nano-ML values* S409c⟩≡ (S411c S373a S3

```
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
```

S409d. ⟨*primitives for nano-ML and* μ*ML* `::` S408e⟩+≡ (S411b) ◁ S409b

```
("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.1.4 Predefined functions

S409e. ⟨*predefined nano-ML functions* S409e⟩≡ S410a ▷

```
(define list1 (x) (cons x '()))
(define bind (x y alist)
  (if (null? alist)
    (list1 (pair x y))
    (if (= x (fst (car alist)))
      (cons (pair x y) (cdr alist))
      (cons (car alist) (bind x y (cdr alist))))))
```

We need a test to see if a variable is bound. When a variable is unbound, we 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.

S410a. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S409e S417d▷

```
(define bound? (x alist)
  (if (null? alist)
    #f
    (if (= x (fst (car alist)))
      #t
      (bound? x (cdr alist)))))
(define find (x alist)
  (if (null? alist)
    (error 'not-found)
    (if (= x (fst (car alist)))
      (snd (car alist))
      (find x (cdr alist)))))
```

R.1.5 Processing definitions: elaboration and evaluation

As in Typed Impcore and Typed μScheme, we process a definition by first elaborating it (which includes inferring its type), then evaluating it. The elaborator and evaluator produce strings that respectively represent type and value. If the value string is nonempty, we print both strings. If definition `d` is not well typed, calling `typdef` raises the `TypeError` exception, and we never call `evaldef`.

S410b. ⟨*definitions of* `basis` *and* `processDef` *for nano-ML* S410b⟩≡ (S406a)

```
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 prints 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`.

R.1.6 The read-eval-print loop

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`, we have a handler for `BugInTypeInference`.

S410c. ⟨*other handlers that catch non-fatal exceptions and pass messages to* `caught` S410c⟩≡

```
| TypeError          msg => caught ("type error <at loc>: " ^ msg)
| BugInTypeInference msg => caught ("bug in type inference: " ^ msg)
```

S411a. ⟨*more handlers for* `atLoc` S411a⟩≡

```
  | e as TypeError _           => raise Located (loc, e)
  | e as BugInTypeInference _ => raise Located (loc, e)
```

R.1.7 Building the initial basis

Given primitives and user code, we calculate type and value environments simultaneously.

S411b. ⟨*implementations of nano-ML primitives and definition of* `initialBasis` S411b⟩≡ (S411c)

`initialBasis : type_env * value env`

```
⟨shared utility functions for building primitives in languages with type inference S408d⟩
⟨utility functions for building nano-ML primitives S409a⟩
val initialBasis =
  let fun addPrim ((name, prim, tau), (Gamma, rho)) =
        ( bindtyscheme (name, generalize (tau, freetyvarsGamma Gamma), Gamma)
        , bind (name, PRIMITIVE prim, rho)
        )
      val primBasis = foldl addPrim (emptyTypeEnv, emptyEnv)
                                (⟨primitives for nano-ML and μML :: S408e⟩
                                 ⟨primitives for nano-ML :: 451a⟩
                                 [])
      val fundefs = ⟨predefined nano-ML functions, as strings (from ⟨predefined nano-ML functions S409e⟩)⟩
      val xdefs   = stringsxdefs ("predefined functions", fundefs)
  in  readEvalPrintWith predefinedFunctionError (xdefs, primBasis, noninteracti
  end
```

R.1.8 Pulling the pieces together

The overall structure of the nano-ML interpreter resembles the structure of the Typed μScheme interpreter, but instead of type checking, we have type inference.

S411c. ⟨*ml.sml* S411c⟩≡

```
⟨exceptions used in languages with type inference S237c⟩
⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization S237a⟩

⟨Hindley-Milner types with named type constructors S408c⟩

⟨abstract syntax and values for nano-ML S405c⟩
⟨utility functions on μScheme, Typed μScheme, and nano-ML values S409c⟩

⟨type inference for nano-ML and μML S405e⟩

⟨lexical analysis and parsing for nano-ML, providing filexdefs and stringsxdefs S412c⟩

⟨evaluation, testing, and the read-eval-print loop for nano-ML S406a⟩

⟨implementations of nano-ML primitives and definition of initialBasis S411b⟩
⟨function runAs, which evaluates standard input given initialBasis S372c⟩
⟨code that looks at command-line arguments and calls runAs to run the interpreter S372d⟩
```

R.2 Printing types and constraints and substitutions

Function types are printed infix, and other constructor applications are printed prefix.

S411d. ⟨*definition of* `typeString` *for Hindley-Milner types* S411d⟩≡ (S408c)

```
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.

S412a. ⟨*definitions of* `constraintString` *and* `untriviate` S412a⟩≡ (S405d)

```
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
```

When we print a true polytype, we make the `forall` explicit, and we show all the quantified variables.[1]

S412b. ⟨*shared utility functions on Hindley-Milner types* S412b⟩≡ (S408c)

```
typeString       : ty          -> string
typeSchemeString : type_scheme -> string
```

```
fun typeSchemeString (FORALL ([], tau)) =
      typeString tau
  | typeSchemeString (FORALL (a's, tau)) =
      "(forall [" ^ spaceSep a's ^ "] " ^ typeString tau ^ ")"
```

R.3 Parsing

S412c. ⟨*lexical analysis and parsing for nano-ML, providing* `filexdefs` *and* `stringsxdefs` S412c⟩≡ (S411c)

⟨*lexical analysis for µScheme and related languages* S373c⟩
⟨*parsers for single µScheme tokens* S374d⟩
⟨*parsers for nano-ML tokens* S413b⟩
⟨*parsers and parser builders for formal parameters and bindings* S375a⟩
⟨*parsers and parser builders for Scheme-like syntax* S375d⟩
⟨*parser builders for typed languages* S395e⟩
⟨*parsers for Hindley-Milner types with named type constructors* S413c⟩
⟨*parsers and* `xdef` *streams for nano-ML* S412d⟩
⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S254c⟩

S412d. ⟨*parsers and* `xdef` *streams for nano-ML* S412d⟩≡ (S412c) S413d ▷

```
exp      : exp parser
exptable : exp parser -> exp parser
```

```
fun exptable exp =
  let val bindings = bindingsOf "(x e)" name exp
      val dbs      = distinctBsIn bindings
      fun letx kind bs exp = LETX (kind, bs, exp)
      val formals = formalsOf "(x1 x2 ...)" name "lambda"
```

[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, it's better to make the `forall`s explicit.

```
   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  <$> dbs "letrec" <*> exp)
      , ("(let* (bindings) body)",   curry3 LETX LETSTAR <$> bindings     <*> exp)
      ⟨rows added to nano-ML's exptable in exercises S413a⟩
      ]
   end

 val exp = fullSchemeExpOf (atomicSchemeExpOf name) exptable
```


S413a. ⟨*rows added to nano-ML's* `exptable` *in exercises* S413a⟩≡ (S412d)

```
 (* add syntactic extensions here, each preceded by a comma *)
```

When parsing a type, we reject anything that looks like an expression.

S413b. ⟨*parsers for nano-ML tokens* S413b⟩≡ (S412c)

```
 val arrow = eqx "->" name
 val name  = sat (fn n => n <> "->") name  (* an arrow is not a name *)
 val tyvar = quote *> (curry op ^ "'" <$> name <?> "type variable (got quote mark
```

S413c. ⟨*parsers for Hindley-Milner types with named type constructors* S413c⟩≡ (S412c)

```
tyvar : string parser
ty    : ty     parser
```

```
 val arrows = arrowsOf CONAPP funtype

 fun ty tokens = (
      TYCON <$> sat (curry op <> "->") any_name
  <|> TYVAR <$> tyvar
  <|> usageParsers [("(forall (tyvars) type)", bracket ("('a ...)", many tyvar) *
      <!> "nested 'forall' type is not a Hindley-Milner type"
  <|> bracket ("constructor application",
               arrows <$> many ty <*>! many (arrow *> many ty))
 ) tokens

 val tyscheme =
       usageParsers [("(forall (tyvars) type)",
                      curry FORALL <$> bracket ("['a ...]", distinctTyvars) <*> t
   <|> curry FORALL [] <$> ty
   <?> "type"
```

S413d. ⟨*parsers and* `xdef` *streams for nano-ML* S412d⟩+≡ (S412c) ◁S412d S414b▷

```
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 <*> 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 S414a⟩
  ]

val xdef  =  TEST <$> testtable
         <|> DEF  <$> deftable
         <|>          xdeftable
         <|> badRight "unexpected right bracket"
         <|> DEF <$> EXP <$> exp
         <?> "definition"
```


S414a. ⟨*rows added to nano-ML's* `xdeftable` *in exercises* S414a⟩≡ (S413d)

```
(* add syntactic extensions here, each preceded by a comma *)
```

S414b. ⟨*parsers and* `xdef` *streams for nano-ML* S412d⟩+≡ (S412c) ◁S413d

```
val xdefstream = interactiveParsedStream (schemeToken, xdef)
```

R.4 Unit testing

S414c. ⟨*definition of* `testIsGood` *for nano-ML* S414c⟩≡ (S406a)

```
⟨definition of skolemTypes for languages with named type constructors S415d⟩
⟨shared definitions of typeSchemeIsAscribable and typeSchemeIsEquivalent S415e⟩
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 S415a⟩
      ⟨definitions of check{Expect,Assert,Error{Checks that use type inference S415a⟩
      ⟨definition of checkTypeChecks using type inference S415c⟩
      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 S378b⟩
      ⟨shared check{Expect,Assert,Error{Passes, which call outcome S246c⟩
      ⟨definitions of check*Type*Passes using type inference S416c⟩
      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
```

S415a. ⟨*definitions of* `check{Expect,Assert,Error{Checks` *that use type inference* S415a⟩≡ (S414c) S415b ▷

```
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]
```

S415b. ⟨*definitions of* `check{Expect,Assert,Error{Checks` *that use type inference* S415a⟩+≡ (S414c) ◁ S415a

```
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"
```

S415c. ⟨*definition of* `checkTypeChecks` *using type inference* S415c⟩≡ (S414c)

```
fun checkTypeChecks form (e, sigma) =
  let val (tau, c) = ty e
      val theta  = solve c
  in  true
  end handle TypeError msg =>
      failtest ["In (", form, " ", expString e, " " ^ typeSchemeString sigma, "),
                msg]
```

R.4.1 Checking types against type schemes

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 implement this idea, I create a supply of *skolem types* that cannot possibly be part of any type in any nano-ML program.

S415d. ⟨*definition of* `skolemTypes` *for languages with named type constructors* S415d⟩≡ (S414c)

```
skolemTypes  : ty stream
```

```
val skolemTypes = streamMap (fn n => TYCON ("skolem type " ^ intString n)) naturals
```

I use skolem types to create an "arbitrary" instance of σ_i. If that instance can be made equal to a fresh instance of σ_g, then σ_g is as general as σ_i.

S415e. ⟨*shared definitions of* `typeSchemeIsAscribable` *and* `typeSchemeIsEquivalent` S415e⟩≡ (S414c) S415f ▷

```
asGeneralAs  : type_scheme * type_scheme -> bool
```

```
fun asGeneralAs (sigma_g, sigma_i as FORALL (a's, tau)) =
  let val theta = bindList (a's, streamTake (length a's, skolemTypes), emptyEnv)
      val skolemized = tysubst theta tau
      val tau_g = freshInstance sigma_g
  in  (solve (tau_g ~ skolemized); true) handle _ => false
  end
```

Two type schemes are equivalent if each is as general as the other. (Notice that equivalent type schemes have the same instances.)

S415f. ⟨*shared definitions of* `typeSchemeIsAscribable` *and* `typeSchemeIsEquivalent` S415e⟩+≡ (S414c) ◁ S415e

```
fun eqTypeScheme (sigma1, sigma2) =
  asGeneralAs (sigma1, sigma2) andalso asGeneralAs (sigma2, sigma1)
```

With asGeneralAs and eqTypeScheme in hand, we can implement the unit tests. The check-type checks to see if the type of e is as general as the type being claimed for e.

S416a. ⟨*shared definitions of* typeSchemeIsAscribable *and* typeSchemeIsEquivalent S415e⟩+≡ (S414c) ◁S41

```
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 check-principal-type checks for equivalence.

S416b. ⟨*shared definitions of* typeSchemeIsAscribable *and* typeSchemeIsEquivalent S415e⟩+≡ (S414c) ◁S41

```
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 *)
```

The implementations compute sigma_e.

S416c. ⟨*definitions of* check*Type*Passes *using type inference* S416c⟩≡ (S414c) S416d ▷

```
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]
```

S416d. ⟨*definitions of* check*Type*Passes *using type inference* S416c⟩+≡ (S414c) ◁S416c S416e ▷

```
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.

S416e. ⟨*definitions of* check*Type*Passes *using type inference* S416c⟩+≡ (S414c) ◁S416d

```
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.4.2 Rendering expressions as strings

S417a. ⟨*definition of* `expString` *for nano-ML and μML* S417a⟩≡ (S405c)

```
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
         | LAMBDA (xs, body) => bracketSpace ["lambda",
                                              bracketSpace xs, expString body]
         ⟨extra cases of expString for μML S417b⟩
  end
```

S417b. ⟨*extra cases of* `expString` *for μML* S417b⟩≡ (S417a)

```
(* this space is filled in by the uML appendix *)
```

S417c. ⟨*definitions of* `defString` *and* `defName` *for nano-ML and μML* S417c⟩≡ (S405c)

```
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 generated automatically⟩
  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 generated automatically⟩
```

R.5 Predefined functions

These predefined functions are identical to what we find in μScheme.

S417d. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S410a S418a▷

```
(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))
```

S418a. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S417d S418b▷

```
(define append (xs ys)
  (if (null? xs)
     ys
     (cons (car xs) (append (cdr xs) ys))))
(define revapp (xs ys)
  (if (null? xs)
     ys
     (revapp (cdr xs) (cons (car xs) ys))))
(define reverse (xs) (revapp xs '()))
```

S418b. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S418a S418c▷

```
(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)))
```

S418c. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S418b S418d▷

```
(define filter (p? xs)
  (if (null? xs)
    '()
    (if (p? (car xs))
      (cons (car xs) (filter p? (cdr xs)))
      (filter p? (cdr xs)))))
```

S418d. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S418c S418e▷

```
(define map (f xs)
  (if (null? xs)
    '()
    (cons (f (car xs)) (map f (cdr xs)))))
```

S418e. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S418d S418f▷

```
(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)))
```

S418f. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S418e S419a▷

```
(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))))
```

S419a. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S418f S419b▷

```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

S419b. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S419a S419c▷

```
(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))))
```


S419c. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S419b S419d▷

```
(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)))
```

S419d. ⟨*predefined nano-ML functions* S409e⟩+≡ ◁S419c

```
(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.6 Cases and code for Chapter 8

μML (Chapter 8) is built on nano-ML, with additional cases for pattern matching and algebraic data types. The following code chunks are placeholders for code that is added in Chapter 8.

S419e. ⟨*extra case for* `typdef` *used only in μML* S419e⟩≡ (449f)

```
(* filled in when implementing uML *)
```

S419f. ⟨*exhaustiveness analysis for μML* S419f⟩≡ (S405e)

```
(* filled in when implementing uML *)
```

Chapter contents

Supporting code for μML

S

S.1 Details

Predefined tuple types

S421a. ⟨*predefined μML types* S421a⟩≡ S421b ▷

```
(data (* * * => *) triple
  [TRIPLE : (forall ['a 'b 'c] ('a 'b 'c -> (triple 'a 'b 'c)))])
```

When defining larger tuples, the notation of the explicit form is a bit much. I shift to the implicit form.

S421b. ⟨*predefined μML types* S421a⟩+≡ ◁ S421a S442g ▷

```
(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.1.1 Interpreter: syntax

S421c. ⟨*forms of* `exp` *carried over from nano-ML* S421c⟩≡ (498a)

```
    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_kind * (name * exp) list * exp
  | LAMBDA     of name list * exp
and let_kind = LET | LETREC | LETSTAR
```

S421d. ⟨*forms of* `def` *carried over from nano-ML* S421d⟩≡ (498b)

```
  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`.

S422a. ⟨*definition of* `unit_test` *for languages with Hindley-Milner types and generated type constructors* S422a⟩≡

```
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
```


The representations defined above are combined with representations from other chapters as follows:

S422b. ⟨*abstract syntax and values for µML* S422b⟩≡ (S433f)

⟨*kinds for typed languages* S425a⟩
⟨*definition of* `tyex` *for µML* S425c⟩
⟨*definition of* `pat`*, for patterns* 498c⟩
⟨*definitions of* `exp` *and* `value` *for µML* 498a⟩
⟨*definition of* `def` *for µML* 498b⟩
⟨*definition of* `implicit_data_def` *for µML* S452a⟩
⟨*definition of* `unit_test` *for languages with Hindley-Milner types and generated type constructors* S422a⟩
⟨*definition of* `xdef` *(shared)* S365b⟩
⟨*definition of* `valueString` *for µML* S448b⟩
⟨*definition of* `patString` *for µML and µHaskell* generated automatically⟩
⟨*definition of* `expString` *for nano-ML and µML* S417a⟩
⟨*definitions of* `defString` *and* `defName` *for nano-ML and µML* S417c⟩
⟨*definition of* `tyexString` *for µML* S449c⟩

S.1.2 Support for type equivalence and generativity

S422c. ⟨`tycon`*,* `freshTycon`*,* `eqTycon`*, and* `tyconString` *for generated type constructors* S422c⟩≡ (S434a) S422d ▷

```
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 define `node`, it prints as `node`. But the second time I define `node`, it prints as `node@{2}`, and so on.

S422d. ⟨`tycon`*,* `freshTycon`*,* `eqTycon`*, and* `tyconString` *for generated type constructors* S422c⟩+≡ (S434a) ◁ S422c S423a ▷

`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
```

[1]The second message is from gcc.

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 special type constructors described in Section C.1.

S423a. ⟨`tycon`, `freshTycon`, `eqTycon`, *and* `tyconString` *for generated type constructors* S422c⟩+≡ (S434a) ◁ S422d

```
local                                          freshTycon : name -> tycon
  val nextIdentity = ref 0
  fun freshIdentity () = !nextIdentity before nextIdentity := !nextIdentity + 2
in
  fun freshTycon t = { identity = freshIdentity(), printName = freshPrintName t }
end
```


S.1.3 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

S423b. ⟨*type constructors built into μML and μHaskell* S423b⟩≡ (S434a)

```
val funtycon  = freshTycon "function"
val argstycon = freshTycon "arguments"
```

The first two type constructors are used to make the `int` and `sym` types.

S423c. ⟨*types built into μML and μHaskell* S423c⟩≡ (S434a)

```
val inttype = TYCON inttycon
val symtype = TYCON symtycon
```

The second two are used to make function types, which we can construct and deconstruct.

S423d. ⟨*code to construct and deconstruct function types for μML* S423d⟩≡ (S434a)

```
                                      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
```

bind	312b
CONAPP	418
type def	498b
emptyEnv	311a
type env	310b
eqTycon	497c
type exp	498a
find	311b
inttycon	497d
NotFound	311b
symtycon	497d
type ty	418
TYCON	418
type tyex	S425c
type tyvar	418

S.1.4 Validation of constructor types in `data` definitions

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, so I can pattern match on them. The shapes are $\tau_1 \times \cdots \times \tau_n \to \tau$, τ, $\forall \alpha_1, \ldots, \alpha_k . \tau_1 \times \cdots \times \tau_n \to \tau$, and $\forall \alpha_1, \ldots, \alpha_k . \tau$.

S423e. ⟨*shared utility functions on Hindley-Milner types* S423e⟩≡ (S434a S408c) S424a ▷

```
                                                         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.

S424a. ⟨*shared utility functions on Hindley-Milner types* S423e⟩+≡ (S434a S408c) ◁ S423e

```
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)
```

Supporting code for μML

S424

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 `validateTypeArguments` 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.

S424b. ⟨*definition of* `validate`, *for the types of the value constructors of* `T` S424b⟩≡ (501b)

```
appliesMu             : ty -> bool
validateTypeArguments : ty -> unit
validateLengths       : tyvar list * kind list -> unit
```

```
fun validate (K, sigma as FORALL (alphas, _), mu, kind) =
  let ⟨definitions of appliesMu and validateTypeArguments S451a⟩
      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 S424c⟩
  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.

S424c. ⟨*validation by case analysis on* `schemeShape shape` *and* `kind` S424c⟩≡ (S424b)

```
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)
          ; validateTypeArguments tau
          )
        else
          ⟨type of K should be desiredType but is sigma S451d⟩
    | (POLY_FUN (alphas, _, result), ARROW (argkinds, _)) =>
        if appliesMu result then
          ( validateLengths (alphas, argkinds)
          ; validateTypeArguments result
          )
        else
          ⟨result type of K should be desiredType but is result S451e⟩
    | _ =>
        ⟨for K, complain that alphas is inconsistent with kind S451c⟩
```


When `implicit-data` is translated into `data`, as long as all the t's elaborate to σ's, each σ satisfies the compatibility judgment $\sigma \preccurlyeq \mu :: \kappa$.

S.1.5 *Translation and kind checking of type syntax*

μML uses the same kind system as Typed μScheme.

S425a. ⟨*kinds for typed languages* S425a⟩≡ (S422b S394b) S425b ▷

```
datatype kind = TYPE                          (* kind of all types *)
              | ARROW of kind list * kind     (* kind of many constructors *)
```

S425b. ⟨*kinds for typed languages* S425a⟩+≡ (S422b S394b) ◁ S425a S449d ▷

```
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')
```

MISPLACED: We begin our tour of syntax with type expressions: a type expression in μML is just like a type expression in Typed μScheme (page 366). 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. The translation transforms syntax t (ML type `tyex`) into a type scheme σ (`type_scheme`). It is described in Section 8.7.2 on page 502.

S425c. ⟨*definition of* `tyex` *for µML* S425c⟩≡ (S422b)

```
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 418 and 498, 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; we just have 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`.

Syntax	*Concept*	*Semantics*
t	Type	τ
α	Type variable	α
T	Type name or constructor	μ
`(`$t_1 \cdots t_n$ `->` 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
`(forall (`$\alpha_1 \cdots \alpha_n$`)` t`)`	Quantified type	$\forall \alpha_1, \ldots, \alpha_n . \tau$

Table S.1: Notational correspondence between type syntax and types

The type theory that specifies `txType` is a conservative extension of theory of kind checking from Typed μScheme (function `kindof` on page 387). 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 I erase the types from environment Δ and I erase the syntax t from the judgment $\Delta \vdash t \rightsquigarrow \tau :: \kappa$, I wind up with Typed μScheme's kind system. (Prove it for yourself in Exercise 31.)

Each clause of `txType` implements the translation rule that corresponds to its syntax. Translation rules (Figure 8.6) extend Typed μScheme's kinding rules. To start, a type name or type variable is looked up in the environment Δ.

S426a. ⟨*translation of μML type syntax into types* S426a⟩≡ (S433f) S426b ▷

```
fun txType (TYNAME t, Delta) =        txType : tyex * (ty * kind) env -> ty * kind
      (find (t, Delta)
       handle NotFound _ => raise TypeError ("unknown type name " ^ t))
  | txType (TYVARX a, Delta) =
      (find (a, Delta)
       handle NotFound _ => raise TypeError ("type variable " ^ a ^ " is not in scope"))
```

Constructor application must be well-kinded.

S426b. ⟨*translation of μML type syntax into types* S426a⟩+≡ (S433f) ◁ S426a S426c ▷

```
  | 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 S453a⟩
             | TYPE =>
                   ⟨type tau is not expecting any arguments S453b⟩
      end
```

A function type may be formed only when the argument and result types have kind `TYPE`.

S426c. ⟨*translation of μML type syntax into types* S426a⟩+≡ (S433f) ◁ S426b S427a ▷

```
  | txType (FUNTYX (txs, tx), Delta) =
      let val tks = map (fn tx => txType (tx, Delta)) txs
          val tk  = txType (tx, Delta)
```

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` $([\alpha_1, \ldots, \alpha_n], \tau)$

Table S.2: Representational correspondence between type syntax and types

```
          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.

S427a. ⟨*translation of μML type syntax into types* S426a⟩+≡ (S433f) ◁S426c S427b▷

```
  | 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 $*$, there is no need to write the kind in the judgment.)

In a type *scheme*, `forall` is permitted. Each type variable is given kind $*$.

$$\frac{\alpha_1, \ldots, \alpha_n \text{ are all distinct} \qquad \Delta\{\alpha_1 \mapsto (\alpha_1, *), \ldots, \alpha_n \mapsto (\alpha_n, *)\} \vdash t \rightsquigarrow \tau :: *}{\Delta \vdash \texttt{(forall (}\alpha_1 \;\cdots\; \alpha_n\texttt{)}\ t\texttt{)} \rightsquigarrow \forall \alpha_1, \ldots, \alpha_n . \tau :: *} \qquad \text{(SCHEMEKINDALL)}$$

The distinctness of $\alpha_1, \ldots, \alpha_n$ is guaranteed by the parser, so no check is required here.

S427b. ⟨*translation of μML type syntax into types* S426a⟩+≡ (S433f) ◁S427a S428a▷

`txTyScheme : tyex * (ty * kind) env -> type_scheme`

```
  fun txTyScheme (FORALLX (alphas, tx), Delta) =
        let val Delta'   = extend (Delta, map (fn a => (a, (TYVAR a, TYPE))) alpha
            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 " ^ kindStr
        end
```

If there's no `forall` in the syntax, a type is also a type scheme (with an empty $\forall$).

$$\frac{\Delta \vdash t \rightsquigarrow \tau :: *}{\Delta \vdash t \rightsquigarrow \forall.\tau :: *} \qquad \text{(SCHEMEKINDMONOTYPE)}$$

S428a. ⟨*translation of µML type syntax into types* S426a⟩+≡ (S433f) ◁S427b

```
| 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.1.6 Operational semantics and evaluation

For syntactic forms other than the `case` and `data` forms, µML shares both operational semantics and code with nano-ML. What's new are the rules for case expressions, pattern matching, and the `data` definition.

The components of the evaluator and read-eval-print loop are organized as follows:

S428b. ⟨*evaluation, testing, and the read-eval-print loop for µML* S428b⟩≡ (S433f)
⟨*definition of* `namedValueString` *for functional bridge languages* S399c⟩
⟨*definitions of* `match` *and* `Doesn'tMatch` 506b⟩
⟨*definitions of* `eval` *and* `evaldef` *for nano-ML and µML* S406b⟩
⟨*definition of* `processDef` *for µML* S430a⟩
⟨*shared definition of* `withHandlers` S371a⟩
⟨*shared unit-testing utilities* S246d⟩
⟨*definition of* `testIsGood` *for µML* S449e⟩
⟨*shared definition of* `processTests` S247b⟩
⟨*shared read-eval-print loop and* `processPredefined` S369a⟩

µML also has special syntax for a value-constructor expression, but it isn't interesting: like a value variable, a value constructor is evaluated by looking it up in the environment:

S428c. ⟨*more alternatives for* `ev` *for nano-ML and µML* S428c⟩≡ (S406b)

```
| ev (VCONX vcon) = find (vcon, rho)
```

S428d. ⟨*utility functions on µML syntax* S428d⟩≡ (S433f)

```
fun isfuntype (FORALLX (_, tau)) = isfuntype tau
  | isfuntype (FUNTYX _)          = true
  | isfuntype _                   = false
```

Extension is an operation we also see in LET forms, but this is the first interpreter in which I write it as a function.

S428e. ⟨*support for names and environments* S428e⟩≡ (S237a) S428f▷

```
fun extend (rho, bindings) =          extend : 'a env * 'a env -> 'a env
  foldr (fn ((x, a), rho) => bind (x, a, rho)) rho bindings
```

Function `disjointUnion` combines environments and checks for duplicate names. If it finds a duplicate name, it raises `DisjointUnionFailed`. This exception can be raised only during type inference, not during evaluation.

S428f. ⟨*support for names and environments* S428e⟩+≡ (S237a) ◁S428e

```
exception DisjointUnionFailed of name     disjointUnion : 'a env list -> 'a env
fun disjointUnion envs =
  let val env = List.concat envs
  in  case duplicatename (map fst env)
        of NONE => env
         | SOME x => raise DisjointUnionFailed x
  end
```

S429a. ⟨*function* `literal`*, to infer the type of a literal constant* **[[adt]]** S429a⟩≡
⟨*definition of function* `pvconType` S429c⟩
⟨*definition of function* `pattype` 510⟩
⟨*definition of function* `choicetype` 509b⟩

Function `extendTypeEnv` takes a `type_env` on the left but a `type_scheme env` on the right.

S429b. ⟨*specialized environments for type schemes* S429b⟩≡ (S434a S408c)

```
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
```


We get the type of a value constructor in the same way as we get the type of a variable: instantiate its type scheme with fresh type variables.

S429c. ⟨*definition of function* `pvconType` S429c⟩≡ (S429a)

```
fun pvconType (K, Gamma) =
  freshInstance (findtyscheme (K, Gamma))
  handle NotFound x => raise TypeError ("no value constructor named " ^ x)
```

S429d. ⟨*more alternatives for* `ty` S429d⟩≡ (449a)

```
| ty (VCONX vcon) =
    let val tau =
          freshInstance (findtyscheme (vcon, Gamma))
          handle NotFound _ => raise TypeError ("no value constructor named " ^ vcon)
    in  (tau, TRIVIAL)
    end
```

S.1.7 *The rest of the interpreter*

What's left is code to process definitions and create the initial basis. I instantiate the general framework introduced in Chapter 5: I say what a basis is and how we process a definition. I also implement the primitives and the predefined types.

A basis for μML

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.

S429e. ⟨*definition of* `basis` *for μML* S429e⟩≡ (S433f)

```
type basis = type_env * (ty * kind) env * value env
```

Processing definitions

As in other interpreters for statically typed languages, `processDef` first elaborates 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. In the formal type system, we delegate to `typdef` using this rule:

$$\frac{\langle d, \Gamma \rangle \to \langle \Gamma' \rangle}{\langle d, \Gamma, \Delta, M \rangle \to \langle \Gamma', \Delta, M \rangle} \qquad \text{(ReuseDefinition)}$$

S430a. ⟨*definition of* `processDef` *for μML* S430a⟩≡ (S428b)

`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 prints interactivity then
              println (valstring ^ " : " ^ tystring)
            else
              ()
      in  (Gamma', Delta, rho')
      end
```

To process a data definition, use `typDataDef` and `evalDataDef`.

S430b. ⟨*typing and evaluation of* `data` *definitions* S430b⟩≡ (S433f)

`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 prints interactivity then
                ⟨print the new type and each of its value constructors S430c⟩
              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.

S430c. ⟨*print the new type and each of its value constructors* S430c⟩≡ (S430b)

```
let val (T, _, _) = dd
    val (mu, _)    = find (T, Delta')
    val (kind, vcon_types) =
      case tystrings of s :: ss => (s, ss)
                      | [] => let exception NoKindString in raise NoKindString end
in  ( println (typeString mu ^ " :: " ^ kind)
    ; ListPair.appEq (fn (K, tau) => println (K ^ " : " ^ tau)) (vcons, vcon_types)
    )
end
```

Building the initial basis: predefined types, primitives, predefined functions

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 has to be 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:

S431a. ⟨*definitions of* emptyBasis, predefinedTypeBasis, booltype, listtype, *and* unittype S431a⟩≡ (S433f) S431b ▷

```
val emptyBasis = (emptyTypeEnv, emptyEnv, emptyEnv)     emptyBasis : basis
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 :: S432b⟩ nil)
```

Next, the predefined types. Internal function process accepts only data definitions, which can be elaborated without type inference. We add primitive values and user code.

S431b. ⟨*definitions of* emptyBasis, predefinedTypeBasis, booltype, listtype, *and* unittype S431a⟩+≡ (S433f) ◁ S431a S

```
val predefinedTypeBasis =                               predefinedTypeBasis : basis
  let val predefinedTypes = ⟨predefined μML types, as strings (from ⟨predefined μML types 474d⟩)⟩
      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
```

The predefinedTypeBasis is used to define booltype, which is used in type inference, which is used in typdef, which is used in processDef. So when predefinedTypeBasis is defined, processDef is not yet available. I therefore define internal function process, which processes only data definitions. Luckily, typDataDef does not require type inference.

The next step is to add the primitive functions.

S431c. ⟨*implementations of μML primitives and definition of* initialBasis S431c⟩≡ (S433f) S431d
⟨*shared utility functions for building primitives in languages with type inference* S408d⟩
⟨*utility functions for building nano-ML primitives* S409a⟩

```
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 :: S443b⟩ ni
  end
```

And the final step is to add the predefined functions. Here we have access to all of type inference and evaluation, in the form of function readEvalPrintWith.

S431d. ⟨*implementations of μML primitives and definition of* initialBasis S431c⟩+≡ (S433f) ◁ S4

```
val initialBasis =
  let val predefinedFuns =
        ⟨predefined μML functions, as strings (from ⟨predefined μML functions 470⟩)⟩
      val xdefs = stringsxdefs ("predefined functions", predefinedFuns)
  in  readEvalPrintWith predefinedFunctionError (xdefs, primFunBasis, noninterac
  end
```

Internal access to predefined types

Types `bool`, `list`, `unit`, and so on are used not only in the basis, but also inside the interpreter: they are used to infer types, to define primitive functions, or both. I extract them from `predefinedTypeBasis`. I also define types `alpha` and `beta`, which are used to write the types of polymorphic primitives.

S432a. ⟨*definitions of* `emptyBasis`, `predefinedTypeBasis`, `booltype`, `listtype`, *and* `unittype` S431a⟩+≡ (S43

```
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
```

Specifications of primitive types and functions

Like Typed μScheme, μML has both primitive types and primitive values. Primitive types `int` and `sym` are bound into the kinding 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.

S432b. ⟨*primitive type constructors for μML* `::` S432b⟩≡ (S431a)

```
("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* `::` S443b⟩. The code defined there is reused, but because μML uses `CONVAL` instead of `BOOLV`, `PAIR`, and `NIL`, we need new versions of some of the ML functions on which the primitives are built.

The first new function we need 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.

S432c. ⟨*utility functions on μML values* S432c⟩≡ S433c ▷

```
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 testEqual = primitiveEquality
```

S433a. ⟨*utility functions on μML values* **[[mcl]]** S433a⟩≡ S433b ▷

```
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 testEqual = primitiveEquality
```

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 parser for literal S-expressions uses `embedList` to convert a list of S-expressions into an S-expression. The nano-ML version (chunk 315c) uses Standard ML value constructors `PAIR` and `NIL`, but the μML version uses μML value constructors `cons` and `'()`.

S433b. ⟨*utility functions on μML values* **[[mcl]]** S433a⟩+≡ ◁ S433a S433d ▷

`embedList : value list -> value`

```
fun embedList []      = CONVAL (PNAME "'()", [])
  | embedList (v::vs) = CONVAL (PNAME "cons", [ref v, ref (embedList vs)])
```

S433c. ⟨*utility functions on μML values* S432c⟩+≡ ◁ S432c S433e ▷

```
fun embedList []      = CONVAL ("'()", [])
  | embedList (v::vs) = CONVAL ("cons", [v, 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.

S433d. ⟨*utility functions on μML values* **[[mcl]]** S433a⟩+≡ ◁ S433b

`projectBool : value -> bool`
`embedBool   : bool  -> value`

```
fun embedBool b = CONVAL (PNAME (if b then "#t" else "#f"), [])
fun projectBool (CONVAL (PNAME "#t", [])) = true
  | projectBool _                         = false
```

S433e. ⟨*utility functions on μML values* S432c⟩+≡ ◁ S433c

```
fun embedBool b = CONVAL (if b then "#t" else "#f", [])
fun projectBool (CONVAL ("#t", [])) = true
  | projectBool _                   = false
```

Pulling the pieces together

The full interpreter shares lots of components with nano-ML.

S433f. ⟨*uml.sml* S433f⟩≡

⟨*exceptions used in languages with type inference* S237c⟩
⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S237a⟩

⟨*Hindley-Milner types with generated type constructors* S434a⟩

⟨*abstract syntax and values for μML* S422b⟩
⟨*utility functions on μML syntax* S428d⟩

```
⟨utility functions on μML values generated automatically⟩

⟨lexical analysis and parsing for μML, providing filexdefs and stringsxdefs S437a⟩

⟨definition of basis for μML S429e⟩
⟨translation of μML type syntax into types S426a⟩
⟨typing and evaluation of data definitions S430b⟩
⟨definitions of emptyBasis, predefinedTypeBasis, booltype, listtype, and unittype S431a⟩
⟨type inference for nano-ML and μML S405e⟩

⟨evaluation, testing, and the read-eval-print loop for μML S428b⟩

⟨implementations of μML primitives and definition of initialBasis S431c⟩
⟨function runAs, which evaluates standard input given initialBasis S372c⟩
⟨code that looks at command-line arguments and calls runAs to run the interpreter S372d⟩
```

Most of the type components are shared with either nano-ML or μHaskell.

S434a. ⟨*Hindley-Milner types with generated type constructors* S434a⟩≡ (S433f)

```
⟨tycon, freshTycon, eqTycon, and tyconString for generated type constructors S422c⟩
⟨representation of Hindley-Milner types 418⟩
⟨sets of free type variables in Hindley-Milner types 442⟩
⟨type constructors built into μML and μHaskell S423b⟩
⟨types built into μML and μHaskell S423c⟩
⟨code to construct and deconstruct function types for μML S423d⟩
⟨definition of typeString for Hindley-Milner types S411d⟩
⟨shared utility functions on Hindley-Milner types S423e⟩
⟨specialized environments for type schemes S429b⟩
⟨extensions that support existential types S434b⟩
```

S.2 Existential types

Before going on with the type theory, here is what we have so far, made concrete in code. First, function asX. Only a function type can be converted to existential. We find the result type by stripping off the function arrow. We then look at the result type's parameters; those are the $\alpha_1, \ldots, \alpha_n$. And whatever original parameters are left over are the $\beta_1, \ldots, \beta_m$.

S434b. ⟨*extensions that support existential types* S434b⟩≡ (S434a) S435a ▷

```
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 _ = let exception GADT in raise GADT end
      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
```

In order to skolemize an existential type, we have to have 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 from scratch, I would prefer to add `SKOLEM_TYPE` to the representation of `ty`, but because I have lots of constraint-solving and type-inference code leftover from nano-ML, I prefer a representation that permits me to reuse that code.)

S435a. ⟨*extensions that support existential types* S434b⟩+≡ (S434a) ◁S434b S436d▷

```
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)
```

Finally, function `pvconType` implements the judgment $\Gamma \vdash_p K : \tau$

S435b. ⟨*definition of function* `pvconType` **[[existentials]]** S435b⟩≡

```
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 (bindList (betas, skolems, emptyEnv))
               in  FORALL (alphas, theta (funtype (args, result)))
               end
  in  freshInstance sigma'
  end handle NotFound x => raise TypeError ("no value constructor named " ^ x)
```

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

The rule is implemented using these representations:

p	e	Γ	Γ'	$\tau \to \tau'$	$C \wedge C'$
`p`	`e`	`Gamma`	`Gamma'`	`ty`	`con`

To find the free skolem types of $\theta\Gamma'$, I look at all the types bound in $\theta\Gamma'$. But to find the free skolem types of $\theta\Gamma$, I need to look only at what θ substitutes for the free type variables of Γ.

S435c. ⟨*check* `p`, `e`, `Gamma'`, `Gamma`, `ty`, *and* `con` *for escaping skolem types* **[[existentials]]** S435c⟩≡

```
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 Gam
    val tySkolems  = typeFreeSkolems  (tysubst theta ty)
    ⟨definitions of skolem functions fail and badType S436c⟩
in  case (inter (patSkolems, tySkolems), inter (patSkolems, envSkolems))
      of (mu :: _,  _) => ⟨fail with skolem escaping into type S436a⟩
       | ([], mu :: _) => ⟨fail with skolem escaping into environment S436b⟩
       | ([], []) => ()
end
```

If $\tau \to \tau'$ has an escaping skolem type, I check τ' first, then τ.

S436a. ⟨*fail with skolem escaping into type* S436a⟩≡ (S435c)

```
(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]
    | _ => let exception ChoiceTypeNotFun in raise ChoiceTypeNotFun end)
```

If the problem is in the environment, I don't provide much help.

S436b. ⟨*fail with skolem escaping into environment* S436b⟩≡ (S435c)

```
fail ["skolem type " ^ tyconString mu ^ " constrains a variable in the environment"]
```

All the failure modes identify the problematic pattern match and raise `TypeError`.

S436c. ⟨*definitions of skolem functions* `fail` *and* `badType` S436c⟩≡ (S435c)

```
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"]
```

I find free skolem types by examining every type constructor. I want only to add a skolem type to an existing set, not to allocate multiple sets, so I begin with a function that can be passed to `foldl`.

S436d. ⟨*extensions that support existential types* S434b⟩+≡ (S434a) ◁S435a S436e▷

```
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
```

Using `addFreeSkolems`, I can find free skolem types in a type, in a set of types, or in a list of type schemes.

S436e. ⟨*extensions that support existential types* S434b⟩+≡ (S434a) ◁S436d S436f▷

```
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)
```

My substitution into Γ' is just good enough for patterns—I know that every type scheme in Γ' is a monotype.

S436f. ⟨*extensions that support existential types* S434b⟩+≡ (S434a) ◁S436e

```
typeEnvSubst : subst -> type_scheme env -> type_scheme env
```

```
fun typeEnvSubst theta Gamma' =
  let fun subst (FORALL ([], tau)) = FORALL ([], tysubst theta tau)
        | subst _ = let exception PolytypeInPattern in raise PolytypeInPattern end
  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.

S436g. ⟨*check* `p`, `e`, `Gamma'`, `Gamma`, `ty`, *and* `con` *for escaping skolem types* S436g⟩≡ (509b)

```
()
```

S437a. ⟨*lexical analysis and parsing for μML, providing* `filexdefs` *and* `stringsxdefs` S437a⟩≡ (S433f)
⟨*lexical analysis for μScheme and related languages* S373c⟩
⟨*parsers for single μScheme tokens* S374d⟩
⟨*parsers for μML tokens* S437d⟩
⟨*parsers for μML value constructors and value variables* S437e⟩
⟨*parsers and parser builders for formal parameters and bindings* S375a⟩
⟨*parsers and parser builders for Scheme-like syntax* S375d⟩
⟨*parser builders for typed languages* S395e⟩
⟨*parsers for Hindley-Milner types with generated type constructors* S437b⟩
⟨*parsers and* `xdef` *streams for μML* S438b⟩
⟨*shared definitions of* `filexdefs` *and* `stringsxdefs` S254c⟩

S.3.1 *Parsing types and kinds*

Parsers for types and kinds are as in Typed μScheme, except the type parser produces a `tyex`, not a `ty`.

S437b. ⟨*parsers for Hindley-Milner types with generated type constructors* S437b⟩≡ (S437a) S437c ▷

```
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 tye
) tokens
```

S437c. ⟨*parsers for Hindley-Milner types with generated type constructors* S437b⟩+≡ (S437a) ◁ S437

```
kind : kind parser
```

```
fun kind tokens = (
  TYPE <$ eqx "*" vvar
  <|> bracket ("arrow kind", curry ARROW <$> many kind <* eqx "=>" vvar <*> kin
) tokens

val kind = kind <?> "kind"
```

S.3.2 *Identifying μML tokens*

From the implementation of μScheme in Appendix O, μML inherits the token parsers `name`, `booltok`, `quote`, and `int`. Type variables are easily recognized. μML has many different kinds of names, and I want to be precise about which sort of name I mean where. So I rename `name` to `any_name`, and I disable `name` by rebinding it to a useless value.

S437d. ⟨*parsers for μML tokens* S437d⟩≡ (S437a)

```
val tyvar = quote *> (curry op ^ "'" <$> name <?> "type variable (got quote mar
val any_name = name
val name = () (* don't use me as a parser *)
```

A token that presents as a name is one of the following: an arrow, a value constructor, a value variable, or a type name. First the predicates:

S437e. ⟨*parsers for μML value constructors and value variables* S437e⟩≡ (S437a) S438a ▷

```
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)
```

And now the parsers. A value constructor may be not only a suitable name but also a Boolean literal or the empty list.

S438a. ⟨*parsers for µML value constructors and value variables* S437e⟩+≡ (S437a) ◁S437e

```
  val arrow = sat (fn n => n = "->") any_name
  val vvar  = sat isVvar any_name
  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") <$> booltok
    in  boolcon <|> sat isVcon any_name <|>
        "'()" <$ quote <* sat isEmptyList (pair <$> left <*> right)
    end
```

S.3.3 *Parsing patterns*

The distinction between value variable and value constructor is most important in patterns.

S438b. ⟨*parsers and* `xdef` *streams for µML* S438b⟩≡ (S437a) S438c ▷

`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.3.4 *Parsing expressions*

Parsing is more elaborate then usual because I provide for two flavors of each binding construct found in nano-ML: the standard flavor, which binds variables, and the "patterns everywhere" flavor, which binds patterns. (The case expression, of course, binds only patterns.) I begin with parsers for formal parameters, which are used to parse both expressions and definitions. The `vvarFormalsIn` parsers 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.

S438c. ⟨*parsers and* `xdef` *streams for µML* S438b⟩+≡ (S437a) ◁S438b S438d ▷

```
vvarFormalsIn : string -> name list parser
patFormals    :              pat list parser
```

```
  val vvarFormalsIn = formalsOf "(x1 x2 ...)" vvar
  val patFormals    = bracket ("(p1 p2 ...)", many pattern)
```

To parse an expression, I provide two sets of parsers, but I provide only the "expression builders" that work with names. Expression builders that work with patterns are left as exercises.

S438d. ⟨*parsers and* `xdef` *streams for µML* S438b⟩+≡ (S437a) ◁S438c S439b ▷

```
  ⟨utility functions that help implement µML's syntactic sugar S441f⟩
  fun exptable exp =
    let (* parsers used in both flavors *)
        val choice   = bracket ("[pattern exp]", pair <$> pattern <*> exp)
```

```
    val letrecBs = distinctBsIn (bindingsOf "(x e)" vvar exp) "letrec"

    (* parsers for bindings to names *)
    val letBs     = distinctBsIn (bindingsOf "(x e)" vvar exp) "let"
    val letstarBs = bindingsOf "(x e)" vvar exp
    val formals   = vvarFormalsIn "lambda"

    (* parsers for bindings to patterns *)
    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 *)

    (* expression builders that expect to bind names *)
    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"

    ⟨μML expression builders that expect to bind patterns S442d⟩
in  ⟨parsers for expressions that begin with keywords S439a⟩
end
```

The parsers that might change are `formals`, `letBs`, and `letstarBs`. The expression-builders that might change are `lambda`, `lambdastar`, and `letx`.

S439a. ⟨*parsers for expressions that begin with keywords* S439a⟩≡ (S438d)

```
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 S443c⟩
  ]
```

With the keyword expressions defined by `exptable`, here are the atomic expressions and the full expressions.

S439b. ⟨*parsers and* `xdef` *streams for μML* S438b⟩+≡ (S437a) ◁S438d S440a▷

```
val atomicExp =  VAR                <$> vvar
             <|> VCONX              <$> vcon
             <|> (LITERAL o NUM)    <$> int
```

`atomicExp : exp parser`	
`exp       : exp parser`	

```
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.3.5 *Parsing definitions*

I begin with the implicit-data definition, which is parsed here and then transformed to a data definition by function makeExplicit.

S440a. ⟨*parsers and* xdef *streams for μML* S438b⟩+≡ (S437a) ◁S439b S440b▷

`implicitData : def parser`

```
⟨definition of makeExplicit, to translate implicit-data to data S452b⟩
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
```

Here is the parser for the true definitions.

S440b. ⟨*parsers and* xdef *streams for μML* S438b⟩+≡ (S437a) ◁S440a S441a▷

`def : def parser`

```
val def =
  let (* parser for binding to names *)
      val formals = vvarFormalsIn "define"

      (* parsers for clausal definitions, a.k.a. define* *)
      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)

      (* definition builders used in all parsers *)
      val Kty = typedFormalOf vcon (kw ":") tyex
      fun data kind name vcons = DATA (name, kind, vcons)

      (* definition builders that expect to bind names *)
      fun define f xs body = DEFINE (f, (xs, body))
      fun definestar _ = ERROR "define* is left as an exercise"

      ⟨μML definition builders that expect to bind patterns generated automatically⟩
  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 <*> exp)
```

```
      , ("(data kind t [vcon : type] ...)", data <$> kind <*> tyname <*> many Kty)
      ]
  end
```

The parser for unit tests.

S441a. ⟨*parsers and* `xdef` *streams for μML* S438b⟩+≡ (S437a) ◁S440b S441b▷

`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 <|> implic
                                                                <|> EXP <$> exp
  ]
```

The parser for other extended definitions.

S441b. ⟨*parsers and* `xdef` *streams for μML* S438b⟩+≡ (S437a) ◁S441a S441d▷

`xdeftable : xdef parser`

```
val xdeftable = usageParsers
  [ ("(use filename)", USE <$> any_name)
  ⟨rows added to μML's xdeftable in exercises S443d⟩
  ]
```

S441c. ⟨*rows added to μML's* `xdeftable` *in exercises* **[[assert-types]]** S441c⟩≡

S441d. ⟨*parsers and* `xdef` *streams for μML* S438b⟩+≡ (S437a) ◁S441b S443a▷

`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.3.6 *Support for syntactic sugar*

Some syntactic transformations need to find a variable that is not free in a given expression. If you have done Exercise 10 on page 332 in Chapter 5, you're close to having the right test. Use that code to complete function `freeIn` here.

S441e. ⟨*utility functions that help implement μML's syntactic sugar* **[[prototype]]** S441e⟩≡

`freeIn : exp -> name -> bool`

```
fun freeIn exp y =
  let fun has_y (CASE (e, choices)) = has_y e orelse (List.exists choice_has_y)
        | 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, here are a variety of helper functions. 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.

S441f. ⟨*utility functions that help implement μML's syntactic sugar* S441f⟩≡ (S438d) S442a▷

`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 => let exception EmptyVarSupply in raise EmptyVarSupply end
```

Function freshVars returns as many fresh variables as there are elements in xs.

S442a. ⟨*utility functions that help implement μML's syntactic sugar* S441f⟩+≡ (S438d) ◁ S441f S442b ▷

```
fun freshVars e xs =                    freshVars : exp -> 'a list -> name list
  streamTake (length xs, streamFilter (not o freeIn e) varsupply)
```


To support pattern matching in lambda, lambda*, and define*, we turn a sequence of names into a single tuple expression, and we turn a sequence of patterns into a single tuple pattern. Function tupleVcon gives the name of the value constructor for a tuple of the same size as the given list.

S442b. ⟨*utility functions that help implement μML's syntactic sugar* S441f⟩+≡ (S438d) ◁ S442a S442c ▷

```
fun tupleVcon xs = case length xs       tupleexp  : name list -> exp
                     of 2 => "PAIR"      tuplepat  : pat  list -> pat
                      | 3 => "TRIPLE"    tupleVcon : 'a   list -> vcon
                      | n => "T" ^ intString n

fun tupleexp [x] = VAR x
  | tupleexp xs  = APPLY (VCONX (tupleVcon xs), map VAR xs)

fun tuplepat [x] = x
  | tuplepat xs  = CONPAT (tupleVcon xs, xs)
```

Function freePatVars finds the free variables in a pattern.

S442c. ⟨*utility functions that help implement μML's syntactic sugar* S441f⟩+≡ (S438d) ◁ S442b

```
                                         freePatVars : pat -> name set
fun freePatVars (PVAR x)         = insert (x, emptyset)
  | freePatVars (WILDCARD)       = emptyset
  | freePatVars (CONPAT (_, ps)) = foldl union emptyset (map freePatVars ps)
```

The rest of the code is for you to write.

S442d. ⟨*μML expression builders that expect to bind patterns* S442d⟩≡ (S438d)

```
(* you can redefine letx, lambda, and lambdastar here *)
```

S442e. ⟨*μML definition builders that expect to bind patterns* **[[prototype]]** S442e⟩≡

```
(* you can redefine 'define' and 'definestar' here *)
```

S442f. ⟨*rows added to μML's* xdeftable *in exercises* **[[prototype]]** S442f⟩≡

```
(* you can add a row for 'val' here *)
```

S.4 S-EXPRESSION READER

This experimental feature of μML reads S-expressions from a file. It is on hold while I decide if every language in the book should get a little library for reading data from files.

An S-expression is a Boolean, symbol, number, or list of S-expressions.

S442g. ⟨*predefined μML types* S421a⟩+≡ ◁ S421b

```
(data * sx
   [Sx.B : (bool -> sx)]
   [Sx.S : (sym  -> sx)]
   [Sx.N : (int  -> sx)]
   [Sx.L : ((list sx)  -> sx)])
```

We read S-expressions using a little parser.

S443a. ⟨*parsers and* `xdef` *streams for μML* S438b⟩+≡ (S437a) ◁S441d

```
                 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 <$>! @@ any_name)
     <|> sxn <$> 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 produce a list of S-expressions stored in a file.

S443b. ⟨*primitives for nano-ML and μML* `::` S443b⟩≡ (S431c S411b)

```
("read", unaryOp (fn (SYM s) =>
                       let val fd = TextIO.openIn s
                             handle _ => raise RuntimeError ("Cannot read file '
                           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)) ::
```

S443c. ⟨*rows added to μML's* `exptable` *in exercises* S443c⟩≡ (S439a)

```
(* you add this bit *)
```

S443d. ⟨*rows added to μML's* `xdeftable` *in exercises* S443d⟩≡ (S441b)

```
(* you add this bit *)
```

S.5 More predefined functions

Some functions are exactly as in μScheme or nano-ML.

S443e. ⟨*predefined μML functions* S443e⟩≡ S443f▷

```
(define and (b c) (if b  c  b))
(define or  (b c) (if b  b  c))
(define not (b)   (if b #f #t))
```

S443f. ⟨*predefined μML functions* S443e⟩+≡ ◁S443e S443g▷

```
(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)))
```

S443g. ⟨*predefined μML functions* S443e⟩+≡ ◁S443f S444a▷

```
(define caar (xs) (car (car xs)))
(define cadr (xs) (car (cdr xs)))
(define cdar (xs) (cdr (car xs)))
```

List functions are simpler with pattern matching. Compare this code with Section R.5 on page S417.

S444a. ⟨*predefined μML functions* S443e⟩+≡ ◁S443g S444b▷

```
(define filter (p? xs)
  (case xs
     ('()   '())
     ((cons y ys)  (if (p? y) (cons y (filter p? ys))
                               (filter p? ys)))))
```


S444b. ⟨*predefined μML functions* S443e⟩+≡ ◁S444a S444c▷

```
(define map (f xs)
  (case xs
     ('() '())
     ((cons y ys) (cons (f y) (map f ys)))))
```

S444c. ⟨*predefined μML functions* S443e⟩+≡ ◁S444b S444d▷

```
(define app (f xs)
  (case xs
     ('() UNIT)
     ((cons y ys) (begin (f y) (app f ys)))))
```

S444d. ⟨*predefined μML functions* S443e⟩+≡ ◁S444c S444e▷

```
(define reverse (xs) (revapp xs '()))
```

S444e. ⟨*predefined μML functions* S443e⟩+≡ ◁S444d S444f▷

```
(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))))
```

S444f. ⟨*predefined μML functions* S443e⟩+≡ ◁S444e S444g▷

```
(define foldr (op zero xs)
  (case xs
     ('() zero)
     ((cons y ys) (op y (foldr op zero ys)))))
(define foldl (op zero xs)
  (case xs
     ('() zero)
     ((cons y ys) (foldl op (op y zero) ys))))
```

S444g. ⟨*predefined μML functions* S443e⟩+≡ ◁S444f S444h▷

```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

S444h. ⟨*predefined μML functions* S443e⟩+≡ ◁S444g S444i▷

```
(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))))
```

S444i. ⟨*predefined μML functions* S443e⟩+≡ ◁S444h S445a▷

```
(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)))
```

S445a. ⟨*predefined μML functions* S443e⟩+≡ ◁S444i S445b▷

```
(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)))
```


S445b. ⟨*predefined μML functions* S443e⟩+≡ ◁S445a

```
(define takewhile (p? xs)
  (case xs
     ('() '())
     ((cons y ys)
        (if (p? y)
            (cons y (takewhile p? ys))
            '())))))
(define dropwhile (p? xs)
  (case xs
     ('() '())
     ((cons y ys)
        (if (p? y)
            (dropwhile p? ys)
            xs))))
```

S.6 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.6.1 Printing stuff using μML

Because μML doesn't have strings, printing complicated things is a pain. But wait! We can code strings as functions. A value of type `printable` encodes a thing that can be printed. Here are a bunch of functions for making and combining printable things. Time pressure prevents me from documenting them.

S445c. ⟨*printers.uml* S445c⟩≡ S446a▷

```
(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 >>wrap      [int int -> (printable -> printable)])
(check-type >>char      [int -> printable])
(check-type ^           [printable printable -> printable])
(check-type >>concat    [(list printable) -> printable])
(check-type >>space-sep [(list printable) -> printable])
(check-type >>comma-sep [(list printable) -> printable])
(check-type >>newline   printable)
(check-type >>space     printable)
(check-type >>parens    [printable -> printable])
```

Here are the implementations.

S446a. ⟨*printers.uml* S445c⟩+≡ ◁S445c

```
(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)))))
(define >>sep (sep)
 (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))))))
(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.6.2 Drawing simple figures in PostScript

I draw circles, disks, and lines for use in PostScript figures.

S446b. ⟨*postscript.uml* S446b⟩≡ S446c▷

```
(use printers.uml)
(check-type ps-draw-circle [int int int -> unit])
(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)))
```

S446c. ⟨*postscript.uml* S446b⟩+≡ ◁S446b S446d▷

```
(val ps-first-line '%!PS-Adobe-1.0)
```

S446d. ⟨*postscript.uml* S446b⟩+≡ ◁S446c

```
(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)))
```

S.7 Drawing red-black trees with dot

This code is used to draw pictures of red-black trees.

S447a. ⟨*dot.uml* S447a⟩≡ S447b ▷

```
(define indent () (begin (printu 32) (printu 32)))
(define printsp (a) (begin (print a) (printu 32)))
```

S447b. ⟨*dot.uml* S447a⟩+≡ ◁ S447a S447c ▷

```
(check-type print-path [(list sym) -> unit])
(define print-path (p)
 (case p
    ('() UNIT)
    ((cons x xs) (begin (print-path xs) (print x)))))
```

S447c. ⟨*dot.uml* S447a⟩+≡ ◁ S447b S447d ▷

```
(check-type print-node-name [(list sym) -> unit])
(define print-node-name (path)
 (begin
  (print 'N)
  (print-path path)
  (printu 32)))
```

S447d. ⟨*dot.uml* S447a⟩+≡ ◁ S447c S447e ▷

```
(check-type dot-empty [(list sym) -> (list sym)])
  ; print an empty node with the given path, return the path

(define dot-empty (path)
 (begin
  (indent)
  (print-node-name path)
  (println '[shape=circle,label="",style=filled,color=black,width=0.15,height=0.15])
  path))
```

S447e. ⟨*dot.uml* S447a⟩+≡ ◁ S447d S447f ▷

```
(check-type dot-edge [(list sym) (list sym) -> unit])
(define dot-edge (p1 p2)
  (begin
    (indent)
    (print-node-name p1)
    (map print '(- >))
    (printu 32)
    (print-node-name p2)
    (printu 10)))
```

S447f. ⟨*dot.uml* S447a⟩+≡ ◁ S447e S448a ▷

```
(check-type dot-node
  (forall ['a] ((list sym) sym 'a (list sym) (list sym) -> (list sym))))

(define dot-node (path color a left right)
 (begin
  (indent)
  (print-node-name path)
  (printu 91) ; left bracket
```

```
    (if (= color 'red)
       (print 'style=filled,fillcolor=red,)
       (print 'color=black,))
    (print 'label=")
    (print a)
    (print '")
    (printu 93) ; right bracket
    (printu 10) ; newline
    (dot-edge path left)
    (dot-edge path right)
    path))
```

S448a. ⟨*dot.uml* S447a⟩+≡ ◁S447f

```
(check-type dot-graph (forall ['a] (('a -> (list sym)) 'a -> unit)))

(define dot-graph (print-tree t)
  (begin
     (printsp 'digraph)
     (printu 123) ; left brace
     (printsp 'edge) (printu 91) (print 'style=solid) (printu 93) (printu 10)
     (print-tree t)
     (printu 125) ; right brace
     (printu 10))) ; newline
```

S.8 Printing values, patterns, types, and kinds

To print a list, we look only at the *name* of a value constructor (we don't have its type). If a user's μML program redefines the cons value constructor, chaos will ensue.

S448b. ⟨*definition of* valueString *for μML* S448b⟩≡ (S422b) S448c ▷

`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>"
```

As in other interpreters, we have a special way of printing applications of cons.

S448c. ⟨*definition of* valueString *for μML* S448b⟩+≡ (S422b) ◁S448b

```
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
```

S448d. ⟨*extra cases of* expString *for μML* S448d⟩≡ (S417a)

```
| 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
```

S449a. ⟨*definition of* `patString` *for μML and μHaskell* S449a⟩≡

```
  fun patString WILDCARD    = "_"
    | patString (PVAR x)    = x
    | patString (CONPAT (vcon, []))   = vcon
    | patString (CONPAT (vcon, pats)) = "(" ^ spaceSep (vcon :: map patString pats
```

S449b. ⟨*definition of* `patString` *for μML and μHaskell*[[**mcl**]] S449b⟩≡

```
  fun patString WILDCARD    = "_"
    | patString (PVAR x)    = x
    | patString (CONPAT (vcon, []))   = vconString vcon
    | patString (CONPAT (vcon, pats)) = "(" ^ spaceSep (vconString vcon :: map pat
```

S449c. ⟨*definition of* `tyexString` *for μML* S449c⟩≡ (S422b)

```
  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 ^ ")"
```

S449d. ⟨*kinds for typed languages* S425a⟩+≡ (S422b S394b) ◁ S425b

```
  fun kindString TYPE = "*"
    | kindString (ARROW (ks, k)) =
        "(" ^ spaceSep (map kindString ks @ ["=>", kindString k]) ^ ")"
```

S.9 Unit testing

Unit testing is as in nano-ML, except that types in the syntax have to be translated.

S449e. ⟨*definition of* `testIsGood` *for μML* S449e⟩≡ (S428b) S453d ▷

```
  ⟨definition of skolemTypes for languages with generated type constructors S450b⟩
  ⟨shared definitions of typeSchemeIsAscribable and typeSchemeIsEquivalent S415e⟩
  fun testIsGood (test, (Gamma, Delta, rho)) =
    let fun ty e = typeof (e, Gamma)
                   handle NotFound x => raise TypeError ("name " ^ x ^ " is not de
        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 S415a⟩
        ⟨definition of checkTypeChecks using type inference S415c⟩

        fun withTranslatedSigma check form (e, sigmax) =
          check (e, txTyScheme (sigmax, Delta))
          handle TypeError msg =>
            failtest ["In (", form, " ", expString e, " ", tyexString sigmax, "),

        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 S450a⟩
    ⟨shared check{Expect,Assert,Error{Passes, which call outcome S246c⟩
    ⟨definitions of check*Type*Passes using type inference S416c⟩

    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
```

A syntactic value is either a literal or a value constructor applied to zero or more syntactic values.

S450a. ⟨asSyntacticValue *for μML* S450a⟩≡ (S449e)

```
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
```

S450b. ⟨*definition of* skolemTypes *for languages with generated type constructors* S450b⟩≡ (S449e)

```
val skolemTypes =
  streamOfEffects (fn () => SOME (TYCON (freshTycon "skolem type")))
```

S.10 Support for datatype definitions

S.10.1 *Cases for elaboration and evaluation of definitions*

In μML, the DATA definition is handled by function processDef (chunk S430a). Functions typdef and evaldef are reused from nano-ML, with these extra cases which should never be executed.

S450c. ⟨*extra case for* typdef *used only in μML* S450c⟩≡ (449f)

```
  | DATA _ => raise InternalError "DATA reached typdef"
```

S450d. ⟨*clause for* evaldef *for datatype definition (μML only)* S450d⟩≡ (S408a)

```
  | evaldef (DATA _, _) = raise InternalError "DATA reached evaldef"
```

S.10.2 *Validation for datatype definitions*

In the chapter, chunk S424b validates definitions of value constructors. Validation uses several auxiliary functions that are defined here.

Function `appliesMu` checks if a type is made by applying type constructor `mu`.

S451a. ⟨*definitions of* `appliesMu` *and* `validateTypeArguments` S451a⟩≡ (S424b) S451b ▷

```
fun appliesMu (CONAPP (tau, _)) = eqType (tau, TYCON mu)
  | appliesMu _ = false
```

Function `validateTypeArguments` checks to make sure that the arguments to a constructor application are distinct type variables.

S451b. ⟨*definitions of* `appliesMu` *and* `validateTypeArguments` S451a⟩+≡ (S424b) ◁ S451a

```
fun validateTypeArguments (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
  | validateTypeArguments (TYCON _) =
      ()  (* happens only when uML is extended with existentials *)
  | validateTypeArguments _ =
      let exception ImpossibleTypeArguments in raise ImpossibleTypeArguments end
```

When validation fails, much of the code that issues error messages is here.

S451c. ⟨*for* `K`*, complain that* `alphas` *is inconsistent with* `kind` S451c⟩≡ (S424 S451f)

```
(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⟩≡ (S424c S451f)

```
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⟩≡ (S424c S451f)

```
raise TypeError ("value constructor " ^ K ^ " should return " ^ desiredType ^
                 ", but it returns type " ^ typeString result)
```

When we have value constructors with existential types, additional validation is needed.

S451f. ⟨*validation by case analysis on* `schemeShape shape` *and* `kind` **[[existentials]]** S451f⟩≡

```
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
            validateTypeArguments 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
            validateTypeArguments result
          else
            ⟨result type of K should be desiredType but is result S451e⟩
      | _ =>
          ⟨for K, complain that alphas is inconsistent with kind S451c⟩
```

S.11 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.

S452a. ⟨*definition of* `implicit_data_def` *for µML* S452a⟩≡ (S422b)

```
  datatype implicit_data_def
    = IMPLICIT_DATA of tyvar list * name * implicit_vcon list
  and implicit_vcon
    = IMPLICIT_VCON of vcon * tyex list
```

The following code translates an implicit data definition into an explicit one.

S452b. ⟨*definition of* `makeExplicit`*, to translate* `implicit-data` *to* `data` S452b⟩≡ (S440a)

```
                                    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.12 Error cases for elaboration of type syntax

Error messages for bad type syntax are issued here.

S453a. ⟨*applied type constructor* `tx` *has the wrong kind* S453a⟩≡ (S426b)

```
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
```


S453b. ⟨*type* `tau` *is not expecting any arguments* S453b⟩≡ (S426b)

```
raise TypeError ("type " ^ typeString tau ^ " is not a type constructor, but it " ^
                 "was applied to " ^ countString taus "other type")
```

S453c. ⟨*definition of* `xdef` *(shared)* **[[assert-types]]** S453c⟩≡

```
  | ASSERT_PTYPE of name * tyex
```

S453d. ⟨*definition of* `testIsGood` *for μML* S449e⟩+≡ (S428b) ◁S449e

```
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
```

Chapter contents

T Supporting code for the Molecule interpreter

T.1 The most exciting parts of the interpreter

Confirm names:

`{ty,comp,mt}subst{Root,Manifest}`

Maybe change `Manifest` to `Abstract` or just `Type`, i.e., name the thing substituted for?

Ideas:

Standard ML	*Molecule*
signature	module type
structure	module
functor	generic module
functor application	specialized module

- "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.

- Key operation: substitute a path for a module identifier. Most familiarly, we substitute for formal parameters. But we might also substitute for the placeholder, when a signature used to seal a module.

T.1.1 Module identifiers and paths

XXX TODO: re-do stamping as in μML. Note: a path in a module-type definition starts with `MODTYPLACEHOLDER`.

S455. ⟨*paths for Molecule* S455⟩≡ (S500b) S494d ▷

`genmodident : name -> modident`

```
type modcon = { printName : name, serial : int }
datatype modident = MODCON of modcon | MODTYPLACEHOLDER of name

⟨definition of function genmodident S494c⟩

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'
```

T.1.2 Types and type equality

S456a. ⟨*definition of* `ty` *for Molecule* S456a⟩≡ (S500b)

```
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'
```

T.1.3 Declarations and module types

Maybe `dec_component` should be `decty`?

A `ENVMOD` has a module identifier only if it is a *top-level* module and has been *elaborated*. MAYBE WHAT WE NEED INSTEAD IS FOR EVERY ENVMOD TO HAVE A PATH?

S456b. ⟨*definition of* `modty` *for Molecule* S456b⟩≡ (S500b)

```
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

type 'a rooted = 'a * path
fun root (_, path) = path
fun rootedMap f (a, path) = (f a, path)

datatype binding
  = ENVVAL     of ty
  | ENVMANTY   of ty
  | ENVMOD     of modty rooted
  | ENVOVLN    of ty list  (* overloaded name *)
  | ENVMODTY   of modty

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.1.4 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`.

S457a. ⟨*type components of module types* S457a⟩≡ (S500c)

```
abstractTypePaths : modty rooted -> path list
```

```
fun abstractTypePaths (MTEXPORTS cs, path) =
      let fun mts (t, COMPABSTY _) = [PDOT (path, t)]
            | mts (x, COMPMOD mt) = abstractTypePaths (mt, PDOT (path, x))
            | mts _ = []
      in  (List.concat o map mts) cs
      end
  | abstractTypePaths (MTALLOF mts, path) =
      (List.concat o map (fn mt => abstractTypePaths (mt, path))) mts
  | abstractTypePaths (MTARROW _, _) = []   (* could be bogus, cf Leroy rule 21 *)
```

S457b. ⟨*invariants of Molecule* S457b⟩≡ (S500c)

```
fun mixedManifestations mt =
  let val path = PNAME (MODTYPLACEHOLDER "invariant checking")
      val manifests = manifestSubsn (mt, path)
      val abstracts = abstractTypePaths (mt, path)
  in  List.exists (hasKey manifests) abstracts
  end
```

MOVE THE SMART CONSTRUCTOR HERE.

T.1.5 Module subtyping

MUST UNDERSTAND LEROY'S SUBSTITUTIONS HERE.

IDEAS:

- Witness to lack of subtype should be keyed by path.
- Error message should tell the whole story, e.g., "context requires that `t` be both `int` and `bool`."
- Try a cheap and cheerful solution to uninhabited intersections, e.g., incompatible manifest types?

S457c. ⟨`implements` *relation, based on* `subtype` *of two module types* S457c⟩≡ (S500c) S459c ▷

```
csubtype : component * component -> unit error
subtype  : modty * modty -> unit error
```

```
infix 1 >>
fun (OK ()) >> c = c
  | (ERROR msg) >> _ = ERROR msg

fun allE []      = OK ()
  | allE (e::es) = e >> allE es

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, _)) =
```

commaSep S239a
countString S238g
csubtype S458b
ERROR S243b
find 311b
hasKey S495c
InternalError S366f
manifestSubsn S458c
type modident S455
MODTYPLACEHOLDER S455
mtsubstRoot S496a
type name 310a
NotFound 311b
OK S243b
type path S455
type path' S455
PDOT S455
PNAME S455
prightmap S522b
whatcomp S507c
|--> S495b

```
                ERROR ("expected a module that takes " ^
                       countString args "parameter" ^ ", but got an exporting module")
            | st (mt, MTALLOF mts') =
                allE (map (fn mt' => st (mt, mt')) mts')
            | st (mt, MTEXPORTS comps') =
                compsSubtype (components mt, comps')
          and components (MTEXPORTS cs) = cs
            | components (MTALLOF mts) = List.concat (map 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 S458b⟩
                      in  csubtype (find (x, comps), supercomp)
                      end
                        handle NotFound y => raise InternalError "missed present component"
                    val missedMsg =
                      if null absent then OK ()
                      else
                        ERROR ("an interface expected some components that are missing: " ^
                               commaSep
                               (map (fn (x, c) => x ^ " (" ^ whatcomp c ^ ")") absent))
                in  allE (map check present) >> missedMsg
                end
      in  st mts
      end
```


S458a. ⟨*no component* x *matching* c' *in context* S458a⟩≡

```
  raise TypeError ("interface calls for " ^ whatcomp c' ^ " called " ^ x ^
                   ", but the implementation does not provide " ^ x)
```

THIS ONE LOOKS GOOD AND IMPORTANT

S458b. ⟨*definition of* csubtype S458b⟩≡ (S457c)

`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' ^ ", but it has type " ^ typeString tau)
    | csubtype (COMPABSTY _, COMPABSTY _) = OK ()  (* XXX really OK? without comparing paths?
    | csubtype (COMPMANTY _, COMPABSTY _) = OK ()  (* XXX likewise? *)
    | csubtype (COMPMANTY tau, COMPMANTY tau') =
        if eqType (tau, tau') then OK ()
        else ERROR ("interface calls for type " ^ x ^ " to manifestly equal " ^
                    typeString tau' ^ ", 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' ^ ", 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' ^
               ", but implementation provides " ^ whatcomp c)
```

NOT CLEAR IF THIS BELONGS HERE OR IN SUPPLEMENT.

S458c. ⟨*module-type realization* S458c⟩≡ (S500c) S459a ▷

`manifestSubsn : modty rooted -> tysubst`

```
  fun manifestSubsn (MTEXPORTS cs, path) =
        let fun mts (x, COMPMANTY tau) = [(PDOT (path, x), tau)]
              | mts (x, COMPMOD mt) = manifestSubsn (mt, PDOT(path, x))
              | mts _ = []
        in  (List.concat o map mts) cs
        end
    | manifestSubsn (MTALLOF mts, path) =
        (List.concat o map (fn mt => manifestSubsn (mt, path))) mts
    | manifestSubsn (MTARROW _, path) = []   (* could be bogus, cf Leroy rule 21 *)
```


REWRITE THIS CODE USING THE LINGO OF SUBSTITUTION!
NOT CLEAR IF THIS BELONGS HERE OR IN SUPPLEMENT.

This is purely a heuristic to get things looking nice. We filter out redundant manifest-type declarations, and we drop any argument that consists only of redundant declarations (or is otherwise empty).

S459a. ⟨*module-type realization* S458c⟩+≡ (S500c) ◁S458c S459b▷

```
  val simpleSyntacticMeet : modty -> modty =
    let val path = PNAME (MODTYPLACEHOLDER "syntactic meet")
        fun filterManifest (prev', []) = rev prev'
          | filterManifest (prev', mt :: mts) =
              let val manifests = manifestSubsn (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', mts)
              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
```

It establishes this invariant: 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.

S459b. ⟨*module-type realization* S458c⟩+≡ (S500c) ◁S459a S497b▷

```
  fun allofAt (mts, path) =  (* smart constructor, rooted module type *)
    let val mt = MTALLOF mts
        val mantypes = manifestSubsn (mt, path)
        val abstypes = abstractTypePaths (mt, path)
    in  if List.exists (hasKey mantypes) abstypes then
          simpleSyntacticMeet (mtsubstManifest mantypes mt)
        else
          mt
    end
```

What's the path for? First argument to `manifestSubsn` and to `abstractTypePaths`. Which means it's used as the prefix to produce the correct substitution, and that's it. So when we have an intersection type, that's the substitution that is used. (Probably not necessary?)

KEY THING! This is my approximation of Leroy's `modtype_match`. Instead of placing type equalities in an environment, I substitute. The ice is getting thin here.

S459c. ⟨`implements` *relation, based on* `subtype` *of two module types* S457c⟩+≡ (S500c) ◁S457c

```
  val mtsubstManifestDebug = fn theta => fn (super, p) =>
```

```
    let val mt' = mtsubstManifest theta super
        val () = app eprint [countString theta "substitution", "\n"]
        val () = app (fn (pi, tau) => app eprint ["   ", pathString pi, " |--> ", typeString
        val () = app eprint ["realized: ", mtString mt', "\n"]

    in  mt'
    end
  fun implements (p : path, submt, supermt) =
   (*   (app eprint ["At ", pathString p,
                     "\n  sub:  ", mtString submt, "\n  sup: ", mtString supermt, "\n"]; id)
    *)
    let val theta = manifestSubsn (submt, p)
        (* val () = app eprint ["substitution ", substString theta, "\n"] *)
    in  subtype (submt, mtsubstManifest theta supermt)  (* XXX need unmixTypes? *)
    end
```


T.1.6 Possible future home: translate path expressions

If we want to use `txpath` in `pathfind`, move it here.

T.1.7 Looking up path expressions

S460. ⟨*path-expression lookup* S460⟩≡ (S500c)

```
pathfind   : pathex * binding env -> binding
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")

  fun notModule (dcl, px) =
    raise TypeError ("looking for a module, but " ^ pathexString px ^
                     " is a " ^ whatdec dcl)
  fun pathfind (PNAME x, Gamma) = find (snd x, Gamma)
    | pathfind (PDOT (path, x), Gamma) =
        let ⟨definition of mtfind S461b⟩
        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 S499c⟩
        end
    | pathfind (PAPPLY (fpx, actualpxs) : pathex, Gamma) =
       ⟨instantiation of module fpx to actualpxs S461a⟩

  fun addloc loc (PNAME x) = PNAME (loc, x)
    | addloc loc (PDOT (path, x)) = PDOT (addloc loc path, x)
    | addloc loc (PAPPLY _) = raise InternalError "application vcon"

  fun vconfind (k, Gamma) = pathfind (addloc ("bogus", ~99) k, Gamma)
```

This is 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 works even if B is itself an arrow type. Uncurrying, it means that when substituting for the first formal parameter, we substitute in all the remaining formal parameters.

S461a. ⟨*instantiation of module* `fpx` *to* `actualpxs` S461a⟩≡ (S460)

```
let fun rootedModtype px = case pathfind (px, Gamma)
                             of ENVMOD (mt, root) => (mt, root)
                              | dec => notModule (dec, px)
    val (fmod, actuals) = (rootedModtype fpx, map rootedModtype actualpxs)
    val (formals, result) = case fmod
                              of (MTARROW fr, _) => fr
                               | _ => ⟨instantiated exporting module fpx S497c⟩
    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 = manifestSubsn (actmt, actroot)
              val () = if true orelse null mtheta then ()
                else app (fn (pi, tau) => app eprint ["manifestly ", pathString pi, " [redacted]-> ", typeSt
              val subst = mtsubstManifest mtheta o mtsubstRoot theta
              (* XXX need to substitute manifest types from the actuals? *)
          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 S497d⟩
          end
      | resty _ = ⟨wrong number of arguments to fpx S497e⟩
in  ENVMOD (resty (formals, actuals, result), PAPPLY (root fmod, map root actual
end
```

S461b. ⟨*definition of* `mtfind` S461b⟩≡ (S460)

`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 component
                  else
                    unimp "value or module component in multiple signatures"
                else
                  case List.filter manifest comps
                    of [comp] => SOME comp
                     | [] => SOME (hd comps)  (* all abstract *)
                     | _ :: _ :: _ =>
```

```
( app (fn c => app eprint ["saw ", ncompString (x, c), "\n"]) comps
;
                       unimp ("manifest-type component " ^ x ^ " in multiple signatures
)
              end)
fun noComponent (path, x, mt) =
  raise TypeError ("module " ^ pathexString path ^ " does not have a component " ^
                 pathexString (PDOT (path, x)) ^ "; its type is " ^ mtString mt)
```

T.1.8 Abstract syntax and values

S462a. ⟨*definitions of* `exp` *and* `value` *for Molecule* S462a⟩≡ (S500b) S499d ▷

`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_kind * (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_kind = LET | LETSTAR
```

The definitions of Molecule are the definitions of nano-ML, plus `DATA`, `OVERLOAD`, and three module-definition forms.

S462b. ⟨*definition of* `def` *for Molecule* S462b⟩≡ (S500b)

`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
```

T.1.9 Type checking for expressions

Here's how operator overloading works:

- 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.

S463a. ⟨*utility functions on Molecule types* S463a⟩≡ (S500c) S463b ▷

```
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")
```

S463b. ⟨*utility functions on Molecule types* S463a⟩+≡ (S500c) ◁ S463a

`resolveOverloaded : name * ty * ty list -> (ty * int) error`

```
fun okOrTypeError (OK a) = a
  | okOrTypeError (ERROR msg) = raise TypeError msg

fun ok a = okOrTypeError a handle _ => raise InternalError "overloaded non-function?"
fun resolveOverloaded (f, argty : ty, tys : ty list) : (ty * int) error =
  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: " ^ separate ("", ", ") (map typeString tys) ^
  in  findAt (tys, 0)
  end
```

S463c. ⟨`typeof` *a Molecule expression* **[[prototype]]** S463c⟩≡

```
fun typeof (e, Gamma) : ty = raise LeftAsExercise "typeof"
```

S463d. ⟨*type of* `CASE (e, choices)` S463d⟩≡

`typeof : exp * binding env -> ty`
`ty : exp -> ty`

```
let fun badChoice n msg =
      raise TypeError ("in choice " ^ intString n ^ " of case expression, " ^ ms

    val tau = typeof (e, Gamma)
    ⟨definition of function patenv for Molecule S464a⟩


    fun choiceRtype (p, e) =
      let val Gamma' = patenv (p, Gamma, tau)
      in  typeof (e, extendEnv (Gamma, Gamma'))
      end

    val rights = map choiceRtype choices
```

eqType	S494e
ERROR	S243b
type error	S243b
FUNTY	S456a
InternalError	S366f
LeftAsExercise	S237a
type modtyx	S456b
type name	310a
OK	S243b
type pat	S500b
type pathex	S455
separate	S239a
type ty	S456a
type tyex	S456a
TypeError	S237b
typeString	S531c
type value	S499d
type vcon	S500b

```
      fun rightsType [] =
            raise TypeError "empty case expression cannot be assigned a type"
        | rightsType (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
      val tau' = rightsType rights

  in  tau'
  end
```

S464a. ⟨*definition of function* patenv *for Molecule* S464a⟩≡ (S463d) S464b ▷

```
  fun extendEnv (Gamma, bindings) =
    let fun add ((x, d), Gamma) = bind (x, d, Gamma)
    in  foldl add Gamma bindings
    end
```

S464b. ⟨*definition of function* patenv *for Molecule* S464a⟩+≡ (S463d) ◁ S464a S464c ▷

```
  fun pvconType (K, Gamma) =
    (case vconfind (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)
```

S464c. ⟨*definition of function* patenv *for Molecule* S464a⟩+≡ (S463d) ◁ S464b

```
  fun patenv (WILDCARD, _, tau) =
        emptyEnv
    | patenv (PVAR x, _, tau) =
        bind (x, ENVVAL tau, emptyEnv)
    | patenv (CONPAT (K, pats), Gamma, tau) =
        let fun badK what tau' =
              raise TypeError ("expected pattern with type " ^ typeString tau ^
                               ", but found value constructor " ^ vconString K ^
                               " with " ^ what ^ " " ^ typeString tau')

            fun patenvs ([], []) = []
              | patenvs (p::ps, tau::taus) = patenv(p, Gamma, tau) :: patenvs(ps, taus)
              | patenvs _ =
                  raise TypeError ("wrong number of arguments to value constructor " ^ vconSt

        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') =>
```

CASE

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash [p_i \;\; e_i] : \tau \to \tau_i, \quad 1 \le i \le n \qquad \tau_1 = \cdots = \tau_n}{\Gamma \vdash \text{CASE}(e, [p_1 \;\; e_1], \ldots, [p_n \;\; e_n]) : \tau_1}$$

$\boxed{\Gamma \vdash e : \tau}$

VCON

$$\frac{\Gamma(K) = \tau}{\Gamma \vdash K : \tau}$$

$\boxed{\Gamma \vdash [p \;\; e] : \tau \to \tau'}$

CHOICE

$$\frac{\Gamma, \Gamma' \vdash p : \tau \qquad \Gamma + \Gamma' \vdash e : \tau'}{\Gamma \vdash [p \;\; e] : \tau \to \tau'}$$

$\boxed{\Gamma, \Gamma' \vdash p : \tau}$

PATVCON

$$\frac{\Gamma \vdash K : \tau_1 \times \cdots \times \tau_k \to \tau \qquad \Gamma, \Gamma'_i \vdash p_i : \tau_i, \quad 1 \le i \le k \qquad \Gamma' = \Gamma'_1 \uplus \cdots \uplus \Gamma'_k}{\Gamma, \Gamma' \vdash (K \; p_1 \; \cdots \; p_k) : \tau}$$

PATBAREVCON

$$\frac{\Gamma \vdash K : \tau}{\Gamma, \{\} \vdash K : \tau}$$

PATWILDCARD

$$\frac{}{\Gamma, \{\} \vdash \text{WILDCARD} : \tau}$$

PATVAR

$$\frac{}{\Gamma, \{x \mapsto \tau\} \vdash x : \tau}$$

Figure T.1: Typing rules for monomorphic case expressions, choices, and patterns


```
                    raise TypeError ("value constructor " ^ vconString K ^ " is applied to " ^
                                     "patterns, but its type " ^ typeString tau' ^
                                     " is not a function type")
          end
```

T.1.10 Type-checking modules: strengthening

Is this the principal type of a module?

S465a. ⟨*principal type of a module* S465a⟩≡ (S500c)

```
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	S459b
COMPABSTY	S456b
COMPMANTY	S456b
COMPMOD	S456b
COMPVAL	S456b
genmodident	S494c
MTALLOF	S456b
MTARROW	S456b
MTEXPORTS	S456b
type path	S455
pathString	S531b
PDOT	S455
PNAME	S455
TYNAME	S456a

T.1.11 Type-checking modules: generativity of top-level definitions

Function `binding` can be used only in a known context—because if the `def` defines a module, we need to know the path for every component.

S465b. ⟨`context` *for a Molecule definition* S465b⟩≡ (S500c)

```
type context
contextDot : context * name -> path
```

```
datatype context
  = TOPLEVEL
  | INMODULE of path

fun contextDot (TOPLEVEL, name) = PNAME (genmodident name) (* XXX key to uniqueness *)
  | contextDot (INMODULE path, name) = PDOT (path, name)

fun contextString TOPLEVEL = "at top level"
  | contextString (INMODULE p) = "in module " ^ pathString p
```


T.1.12 Type-checking definitions

Type-checking a definition extends the environment. But because definitions nest, we structure things a bit differently. This is why we have `binding`. So when we get a definition, we turn it into a named `binding`. The binding gets added to the environment in `elabd`. Among other benefits, this structure makes it easier to allow certain definition forms at top level only.

S466a. ⟨*elaborate a Molecule definition* S466a⟩≡ (S500c) S466b ▷

```
fun declarableResponse c =
      case c
        of ENVMODTY mt => mtString mt
         | ENVVAL tau => typeString tau
         | ENVMANTY _ => "manifest type"
         | ENVMOD (mt, _) => mtString mt
         | ENVOVLN _ => "overloaded name"
```

S466b. ⟨*elaborate a Molecule definition* S466a⟩+≡ (S500c) ◁ S466a S466c ▷

```
printStrings : string list -> value_printer
defResponse : name * binding -> value_printer
```

```
fun printStrings ss _ vs =
  app print ss
type value_printer = (name -> ty -> value -> unit) -> value list -> unit

fun printMt what m how mt = printStrings [what, " ", m, " ", how, " ", mtString mt]

fun defResponse (x, c) =
  case c
    of ENVVAL tau =>
         (fn printfun => fn [v] => (printfun x tau v; app print [" : ", typeString tau])
                          | _ => raise InternalError "value count for val definition")
     | ENVMANTY tau =>
         let val expansion = typeString tau
         in  if x = expansion then
               printStrings ["abstract type ", x]
             else
               printStrings ["type ", x, " = ", typeString 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"
```

S466c. ⟨*elaborate a Molecule definition* S466a⟩+≡ (S500c) ◁ S466b S467a ▷

```
defPrinter : baredef * binding env -> value_printer
```

```
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

fun defPrinter (d, Gamma) =
      let val x = defName d
      in  defResponse (x, find (x, Gamma))
          handle NotFound _ => raise InternalError "defName not found"
      end
```


S467a. ⟨*elaborate a Molecule definition* S466a⟩+≡ (S500c) ◁S466c S467b▷

```
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)
```

S467b. ⟨*elaborate a Molecule definition* S466a⟩+≡ (S500c) ◁S467a

```
elabd : baredef * context * binding env -> (name * binding) list
```

```
⟨more overloading things S470c⟩
fun elabd (d : baredef, context, Gamma) =
  let fun toplevel what =
        case context
          of TOPLEVEL => id
           | _ => raise TypeError (what ^ " cannot appear " ^ contextString cont
      ⟨new definition of mtypeof S468⟩
  in  case d
        of EXP e => toplevel ("an expression (like " ^ expString e ^ ")")
                        (elabd (VAL ("it", e), context, Gamma))
         | MODULETYPE (T, mtx) =>
             let val mt = elabmt ((mtx, PNAME (MODTYPLACEHOLDER T)), Gamma)
             in  toplevel ("a module type (like " ^ T ^ ")")
                 [(T, ENVMODTY mt)]
             end
         | MODULE (name, mx) =>
             let val root = contextDot (context, name)
                 val mt   = mtypeof ((mx, root), Gamma)
             in  [(name, ENVMOD (mt, root))]
             end
         | 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)))
                 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), (
                 val mt = MTARROW (arrowtype (idformals, Gamma))
             in  [(f, ENVMOD (mt, fpath))]
             end
```

```
        | QNAME px => toplevel ("a qualified name (like " ^ pathexString px ^ ")")
                       (elabd (EXP (VAR px), context, Gamma))
        | DEFINE (name, tau, lambda as (formals, body)) =>
            let val funty = FUNTY (map (fn (n, ty) => ty) formals, tau)
            in  elabd (VALREC (name, funty, LAMBDA lambda), context, Gamma)
            end
        | VAL (x, e) =>
            let val tau = typeof (e, Gamma)
            in  [(x, ENVVAL tau)]
            end
        | 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
        | TYPE (t, tx) =>
            let val tau = elabty (tx, Gamma)
            in  [(t, ENVMANTY tau)]
            end
        | DATA dd => elabDataDef (dd, context, Gamma)
        | OVERLOAD ovl => overloadBindings (ovl, Gamma)
  end
```

WILL WANT TO ADD A CONTEXT TO IDENTIFY THE MODULE TO `subtypeError`.

S468. ⟨*new definition of* `mtypeof` S468⟩≡ (S467b)

```
type value_printer
mtypeof : moddef rooted * binding env -> modty
```

```
fun mtypeof ((m, path), Gamma) =
  let fun ty (MPATH p) = strengthen (findModule (p, Gamma), txpath (p, Gamma))
                           (* YYY only use of txpath --- move it? *)
        | ty (MPATHSEALED (mtx, p)) = sealed (mtx, ty (MPATH p))
        | ty (MUNSEALED defs)       = principal defs
        | ty (MSEALED (mtx, defs))  = sealed (mtx, principal defs)
      and sealed (mtx, mt') =
            let val mt = elabmt ((mtx, path), Gamma)
            in  case implements (path, mt', mt)
                  of OK () => mt
                   | ERROR msg => raise TypeError msg
            end
      and principal ds = MTEXPORTS (elabdefs (ds, INMODULE path, Gamma))
      and elabdefs ([],    c, Gamma) = [] : (name * component) list
        | elabdefs ((loc, d) :: ds, c, Gamma) =
            let val bindings = atLoc loc elabd (d, c, Gamma)
                val comps'   = List.mapPartial asComponent bindings
                val Gamma'   = Gamma <+> bindings
                val comps''  = elabdefs (ds, c, Gamma')
                ⟨definition of asUnique S469a⟩
            in  List.mapPartial (asUnique comps'') comps' @ comps''
            end
  in  ty m
  end
```

S469a. ⟨*definition of* asUnique S469a⟩≡ (S468)

```
fun asUnique following (x, c : component) =
  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)
```

Elaborating definitions

S469b. ⟨*elaboration and evaluation of* data *definitions for Molecule* S469b⟩≡ (S500c) S469c ▷

```
elabDataDef : data_def * context * binding env -> (name * binding) list
```

```
fun elabDataDef ((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 S534a⟩
        | validate (K, tau') =
            if eqType (tau', tau) then
              ()
            else
              ⟨type of K should be tau but is tau' S534b⟩
      val ()      = app validate Ktaus
  in  (* thin ice here: the type component should be abstract? *)
      (T, ENVMANTY tau) :: map (fn (K, tau) => (K, ENVVAL tau)) Ktaus
  end
```

S469c. ⟨*elaboration and evaluation of* data *definitions for Molecule* S469b⟩+≡ (S500c) ◁ S469b S469d ▷

```
fun ddString (_, COMPMANTY _) = "*"  (* paper over the thin ice *)
  | ddString (_, COMPVAL tau) = typeString tau
  | ddString _ = raise InternalError "component of algebraic data type"
```

N.B. Duplicates DATA case in defexps XXX.

S469d. ⟨*elaboration and evaluation of* data *definitions for Molecule* S469b⟩+≡ (S500c) ◁ S469c S470a ▷

```
evalDataDef : data_def * value ref env -> value ref env * string list
```

```
fun evalDataDef ((_, typed_vcons), rho) =
  let fun isfuntype (FUNTY _)         = true
        | isfuntype _                 = false
      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
```

S470a. ⟨*elaboration and evaluation of* data *definitions for Molecule* S469b⟩+≡ (S500c) ◁S469d

processDataDef : data_def * basis * interactivity -> basis

```
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"

type basis = binding env * value ref env
fun processDataDef (dd, (Gamma, rho), interactivity) =
  let val bindings      = elabDataDef (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 tystrings = map ddString comps
      val _ = if prints interactivity then
                ⟨print the new type and each of its value constructors for Molecule S470b⟩
              else
                ()
  in  (Gamma', rho')
  end
```

Supporting code for Molecule

S470

S470b. ⟨*print the new type and each of its value constructors for Molecule* S470b⟩≡ (S470a)

```
let val (T, _) = dd
    val tau = (case find (T, Gamma')
                 of ENVMANTY tau => tau
                  | _ => raise Match)
              handle _ => raise InternalError "datatype is not a type"
    val (kind, vcon_types) =
      case tystrings of s :: ss => (s, ss)
                      | [] => let exception NoKindString in raise NoKindString end
in  ( println (typeString tau ^ " :: " ^ kind)
    ; ListPair.appEq (fn (K, tau) => println (K ^ " : " ^ tau)) (vcons, vcon_types)
    )
end
```

S470c. ⟨*more overloading things* S470c⟩≡ (S467b)

```
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 S471d⟩
      val x = plast p

      val currentTypes =
        (case find (x, Gamma)
           of ENVOVLN vals => vals
            | _ => []) handle NotFound _ => []
  in  (x, ENVOVLN (tau :: currentTypes))
  end

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
```

S471a. ⟨*definitions of* basis *and* processDef *for Molecule* S471a⟩≡ (S501b) S471e ▷

```
  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 S471d⟩
              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 currentVals =
                if null currentTypes then Array.fromList []
                else case find (x, rho)
                       of ref (ARRAY a) => a
                        | _ => raise BugInTypeChecking "overloaded name is not ARR
              val v = evalpath (p, rho)
              val newVals = Array.tabulate (1 + Array.length currentVals,
                                            fn 0 => v | i => Array.sub (currentVal
              *****)
              val newVals = extendOverloadTable (x, evalpath (p, rho), rho)
              val rho' = bind (x, ref (ARRAY newVals), rho)

              val _ = if prints interactivity then
                        app print ["overloaded ", x, " : ", typeString tau, "\n"]
                      else
                        ()
          in  (Gamma', rho')
          end
    in  foldl next (Gamma, rho) ps
    end
```

S471b. ⟨*no overload;* p *hasn't any args* S471b⟩≡

```
  raise TypeError ("function " ^ pathexString p ^ " cannot be overloaded " ^
                   "because it does not take any arguments")
```

S471c. ⟨*no overload;* p *isn't a function* S471c⟩≡

```
  raise TypeError ("value " ^ pathexString p ^ " cannot be overloaded " ^
                   "because it is not a function")
```

S471d. ⟨*can't overload a* c S471d⟩≡ (S470c 471a)

```
  raise TypeError ("only functions can be overloaded, but " ^ whatdec c ^ " " ^
                   pathexString p ^ " is not a function")
```

S471e. ⟨*definitions of* basis *and* processDef *for Molecule* S471a⟩+≡ (S501b) ◁ S471a

```
                                    processDef : def * basis * interactivity -> basis
```

```
  type basis = binding env * value ref env
  fun defmarker (MODULETYPE _) = " = "
    | defmarker (DATA _)       = ""
```

```
  | defmarker _              = " : "

fun processDef ((loc, DATA dd), (Gamma, rho), interactivity) =
      atLoc loc processDataDef (dd, (Gamma, rho), interactivity)
  | processDef ((loc, QNAME px), (Gamma, rho), interactivity) =
      let val c = pathfind (px, Gamma)
          val x = pathexString px
          val respond = println o concat
          fun typeResponse ty = if x = ty then ["abstract type ", x]
                                else ["type ", x, " = ", ty]
          fun response (ENVVAL _) = raise InternalError "ENVVAL reached response"
            | response (ENVMANTY tau)                 = typeResponse(typeString tau)
            | response (ENVMOD (mt as MTARROW _, _)) = ["generic module ", x, " : ", mtStri
            | response (ENVMOD (mt, _))              = ["module ", x, " : ", mtString mt]
            | response (ENVMODTY mt)                 = ["module type ", x, " = ", mtString
            | response (ENVOVLN []) = raise InternalError "empty overloaded name"
            | response (ENVOVLN (tau :: taus)) =
                "overloaded " :: x :: " : " :: typeString tau ::
                map (fn t => "\n           " ^ x ^ " : " ^ typeString t) taus

      val _ = if prints interactivity then
                case c
                  of ENVVAL _ =>
                       ignore (processDef ((loc, EXP (VAR px)), (Gamma, rho), interactivity
                   | _ =>
                       respond (response c)
              else
                ()
      in  (Gamma, rho)
      end
  | processDef ((loc, OVERLOAD ps), (Gamma, rho), interactivity) =
      atLoc loc processOverloading (ps, (Gamma, rho), interactivity)
  | processDef ((loc, d), (Gamma, rho), interactivity) =
     (* (app (fn (x, c) => app print [x, " is ", whatcomp c, "\n"]) Gamma; id) *)
      let val bindings   = atLoc loc elabd (d, TOPLEVEL, Gamma)
          val Gamma      = Gamma <+> bindings
          val printer    = defPrinter (d, Gamma)
          val (rho, vs)  = atLoc loc evaldef (d, rho)

          fun callPrintExp i v =
            APPLY (VAR (PNAME (loc, "print")), [LITERAL v], ref i)

          fun printfun x tau v =
            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 prints interactivity then
```

```
                   (printer printfun vs; print "\n")
                 else
                   ()
      in  (Gamma, rho)
      end
```

T.2 Predefined modules and module types

(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))

S473a. ⟨*definition of module* `Char` S473a⟩≡

```
(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 569a⟩
  (define int new ([n : int]) n)
  (val semicolon    59)
  (val quotemark    39)
  (val left-round    40)
  (val right-round   41)
  (val left-curly   123)
  (val right-curly  125)
  (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.2.1 Unused predefined module types

In addition to the ARRAY module type defined in chunk 539b, Molecule defines

S473b. ⟨*Molecule's predefined module types* S473b⟩≡ (S475a) S474a ▷

```
(module-type PRINTS
   (exports [abstype t]
            [print : (t -> unit)]
            [println : (t -> unit)]))
```

S474a. ⟨*Molecule's predefined module types* S473b⟩+≡ (S475a) ◁ S473b S474b ▷

```
(module-type BOOL
   (exports [abstype t]
            [#f : t]
            [#t : t]))
 ;;;; omitted: and, or, not, similar?, copy, print, println
```

S474b. ⟨*Molecule's predefined module types* S473b⟩+≡ (S475a) ◁ S474a S474c ▷


```
(module-type SYM
   (exports [abstype t]
            [=  : (t t -> Bool.t)]
            [!= : (t t -> Bool.t)]))
 ;;;; omitted: hash, similar?, copy, print, println
```

S474c. ⟨*Molecule's predefined module types* S473b⟩+≡ (S475a) ◁ S474b

```
(module-type ORDER
  (exports [abstype t]
           [LESS : t]
           [EQUAL : t]
           [GREATER : t]))

(module [Order : ORDER]
  (data t
    [LESS : t]
    [EQUAL : t]
    [GREATER : t]))

(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)]))

(generic-module [Relations : ([M : (exports [abstype t]
                                            [compare : (t t -> Order.t)])]
                               --m-> (allof RELATIONS
                                            (exports [type t M.t])))]
  (type t M.t)
  (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]))
  (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]))
  (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])))
```


S475a. ⟨*predefined Molecule types, functions, and modules* S475a⟩≡ S475b ▷
  ⟨*Molecule's predefined module types* S473b⟩

T.2.2 *Resizeable arrays*

S475b. ⟨*predefined Molecule types, functions, and modules* S475a⟩+≡ ◁ S475a S491a ▷
  ⟨*arraylist.mcl* S475c⟩

S475c. ⟨*arraylist.mcl* S475c⟩≡ (S475b)

```
(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)

   (define t from ([i : int])
     (Rep.make (U.new 3) i 0 0))

   (define int size ([a : t]) (Rep.population a))

   (define bool in-bounds? ([a : t] [i : int])
     (if (>= i (Rep.low-index a))
         (< (- i (Rep.low-index a)) (Rep.population a))
         #f))

   (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)))

   (define int lo     ([a : t]) (Rep.low-index a))
   (define int nexthi ([a : t]) (+ (Rep.low-index a) (Rep.population a)))
```

```
(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))))

(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)))

(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))

(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)))

(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)))


(define unit setlo ([a : t] [i : int])
  (Rep.set-low-index! a i))

)
```

T.3 Implementations of Molecule's primitive modules

T.3.1 *Molecule's arrays*

T.3.2 *Conversion between ML functions and Molecule functions*

μScheme has 20 primitive functions. Moleculehas over 140 primitive functions. Defining that many functions to operate *directly* on Molecule values would be a ton of work. Instead, I do it *indirectly*: I write primitive functions that manipulate native ML values, then wrap those functions to their arguments are converted from Molecule values to ML values, and their results are converted from ML values back to Molecule values. The technique is useful for writing interpreters in any language that is statically typed and has higher-order functions; the details can be found in one of my papers (Ramsey 2011).

At bottom is the idea of an *embedding/projection pair*.

S477a. ⟨*conversion between ML values and Molecule values* S477a⟩≡ S477b ▷

```
type ('a, 'b) ep = { embed : 'a -> 'b, project : 'b -> 'a }
```

We typically embed ML results into Molecule values, and we project Molecule values into ML arguments.

S477b. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁ S477a S477c ▷

```
type 'a map = ('a, value) ep

fun project { embed = e, project = p } = p
fun embed   { embed = e, project = p } = e
```

```
project : 'a map -> value -> 'a
embed   : 'a map -> 'a -> value
```

Given an ML type that is used in the interpreter, I can define an embedding/projection pair for that type. I choose types that I know can be embedded, so embedding always succeeds. But projection need not succeed; for example, a Molecule Boolean can't be projected into an ML record. If the type checker is written correctly, such a projection will never be attempted. If a bad projection is attempted anyway, I raise the exception `BugInTypeChecking`.

S477c. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁ S477b S477d ▷

```
fun badRep what =
  raise BugInTypeChecking ("bad representation of " ^ what)

val bool  = { embed = BOOLV, project = projectBool }
val int   = { embed = NUM, project = fn NUM n => n | _ => badRep "int" }
val sym   = { embed = SYM, project = fn SYM n => n | _ => badRep "sym" }
val value = { embed = id, project = id }
val nullmap = { embed = fn _ => NIL, project = fn NIL => () | _ => badRep "null" }
```

```
bool  : bool   map
int   : int    map
sym   : string map
value : value  map
```

Here are maps for arrays, records, sums, and iterators.

S477d. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁ S477c S478a ▷

```
val marray =
  { embed = MARRAY, project = fn MARRAY r => r | _ => badRep "mutable array" }
val iarray =
  { embed = IARRAY, project = fn IARRAY r => r | _ => badRep "immutable array" }

val mrecord =
  { embed = MRECORD, project = fn MRECORD r => r | _ => badRep "mutable record" }
val irecord =
  { embed = IRECORD, project = fn IRECORD r => r | v => badRep "immutable record" }

val moneof =
  { embed = MONEOF, project = fn MONEOF r => r | _ => badRep "mutable oneof" }
val ioneof =
  { embed = IONEOF, project = fn IONEOF r => r | _ => badRep "immutable oneof" }
```

The goal is to define primitive operations. Since a primitive operation is represented as an ML function of type `value list -> value list`, I define a `mapf` for such functions.

S478a. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S477d S478b▷

```
type 'a mapf = ('a, value list -> value list) ep
```

The most important conversion function is the one that adds another argument to a Molecule function. The `**->` operation converts between curried ML functions and uncurried Molecule functions. It builds an embedding/projection pair inductively from `firstarg`, which is an embedding/projection pair for the first argument, and from `lastargs`, which is an embedding/projection pair for a function that takes one less argument. To build `firstarg **-> lastargs`, we need an embedding (`apply`) and a projection (`unapply`).

S478b. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S478a S478c▷

```
                    **->    : 'a map * 'b mapf -> ('a -> 'b) mapf
                    apply   : ('a -> 'b) -> (value list -> value list)
infixr 1 **->       unapply : (value list -> value list) -> ('a -> 'b)
fun (firstarg : 'a map) **-> (lastargs : 'b mapf) : ('a -> 'b) mapf =
  let fun apply (f : 'a -> 'b) = fn actuals =>
        let val (v, vs) =
              case actuals
                of v :: vs => (v, vs)
                 | [] => raise InternalError
                            "not enough arguments to primitive function"
            val f_v = f (project firstarg v)
        in  embed lastargs f_v vs
        end
      fun unapply (f_clu : value list -> value list) =
        fn (v : 'a) => project lastargs (fn vs => f_clu (embed firstarg v :: vs))
  in  { embed = apply, project = unapply }
  end
```

The base case for the conversion of functions is an embedding/projection pair for a function that takes no arguments and returns some results. In Molecule, it is possible to return a *list* of results, but in ML, it is not. If a ML function wants to return multiple results, it must wrap them in a tuple, and if the function wants to return zero results, it must return the empty tuple. To deal with this mismatch in languages, the base case for conversion of a function requires conversions between the ML return type `'a` and the Molecule return type `value list`.

S478c. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S478b S478d▷

```
              results : ('a -> value list) -> (value list -> 'a) -> 'a mapf
fun results a_to_values a_of_values =
  { embed   = (fn (a:'a) => fn clu_args => a_to_values a)
  , project = (fn f_clu => a_of_values (f_clu []) : 'a)
  }
```

What the `results` and `**->` functions do is build up a conversion by building a list of the arguments that the function expects. The map from `results` acts like a function of no arguments, and each `**->` acts like a `cons` operation, adding another argument. (That's why `**->` is declared to associate to the right.) So an integer comparison function, for example, can be mapped using the map `int **-> int **-> results bool`. A single result is such a common case that I define some convenience functions just for that case.

S478d. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S478c S479a▷

```
                            result : 'a map -> 'a mapf
                            *->>   : 'a map * 'b map -> ('a -> 'b) mapf
fun take1 [v] = v
```

```
  | take1 _ = raise InternalError "wrong number of results from primitive"

fun result r = results (fn v => [embed r v]) (fn vs => project r (take1 vs))
infixr 1 **->>
fun t **->> t' = t **-> result t'
```

Functions are also values, so to get from a `mapf`, which works with ML functions of type `value list -> value list`, to a `map`, which works with ML values of type `value`, I define `func`. Embedding is done with `PRIMFUN`, but to project, I have to handle not only primitive functions but also closures.

S479a. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S478d S479b▷

`func : 'a mapf -> 'a map`

```
fun func (arrow : 'a mapf) : ('a map) =
 { embed   = PRIMFUN        o embed arrow
 , project = #project arrow o primitiveOfValue
 }
and primitiveOfValue (PRIMFUN f) = f
  | primitiveOfValue (CLOSURE ((xs, body), rho)) =
      (fn vs => case runStmt (body, bindList (xs, map (ref o SOME) vs, rho), NONE)
                  of RETURNS results => results
                   | TERMINATES => []
                   | _ => raise InternalError "closure executescontrol operator")
  | primitiveOfValue _ = badRep "function"
```

Most specifications I write will have `mapf` types, but I want to embed and project primitive operations as values. I use `efunc` and `pfunc`.

S479b. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S479a S479c▷

`efunc : 'a mapf -> 'a -> value`
`pfunc : 'a mapf -> value -> 'a`

```
fun efunc tyspec f = embed   (func tyspec) f
fun pfunc tyspec v = project (func tyspec) v
```

And to implement a `XRECORD`, which puts its operation in a mutable reference cell, I want to embed each primitive function into a reference cell.

S479c. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S479b S479d▷

`efuncr : 'a mapf -> 'a -> value ref`

```
fun efuncr tyspec f = ref (efunc tyspec f)
```

A Molecule function might not return any values, but an ML function always has to return something. I therefore project a no-value Molecule function into an ML function that returns the empty tuple, which has ML type `unit`.

S479d. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S479c S479e▷

`unit : unit mapf`

```
val unit = results (fn () => []) (fn _ => ()) : unit mapf
```

CLU has a handful of primitive iterators, and I define similar machinery. Fortunately, I need only to embed ML functions as Molecule-iterators; I never need to project a Molecule iterator as an ML function. So the machinery is simple.

S479e. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S479d S480a▷

```
type 'a mapi
iterator : (loop_body -> behavior) mapi
*->* : 'a map * 'b mapi -> ('a -> 'b) mapi
eiterr : 'a mapi -> 'a -> value ref
```

```
infixr **->*
type 'a mapi = 'a -> (value list * loop_body -> behavior)
fun iterator prim ([],    yc) = prim yc
  | iterator prim (_::_, yc) = raise InternalError "too many args to iter"

fun (a **->* f) prim (v::vs, yc) = f (prim (project a v)) (vs, yc)
  | (a **->* f) prim ([],     yc) = raise InternalError "too few args to iter"

fun eiterr imap = ref o PRIMITER o imap
```

Here are convenience functions for two recurring types: the type of a copy operation and the type of a comparison operation.

S480a. ⟨*conversion between ML values and Molecule values* S477a⟩+≡ ◁S479e

```
fun copyOf       tau = tau **->> tau
fun comparisonOf tau = tau **-> tau **->> bool
```

T.3.3 *Utilities for equality, similarity, copying, and printing*

Types like `int`, `bool`, `sym`, and `null` can be tested for equality by testing equality of their ML representations. And because they are immutable, they can be "copied" by the identity function.

S480b. ⟨*functions that build operations for equality, similarity, copying, and printing* S480b⟩≡ S480c ▷

```
fun equalityOps tyspec =
  [ ("=",        efuncr (comparisonOf tyspec) (curry op =))
  , ("!=",       efuncr (comparisonOf tyspec) (curry (not o op =)))
  , ("similar?", efuncr (comparisonOf tyspec) (curry op =))
  , ("copy",     efuncr (copyOf tyspec)       id)
  ]
```

Arrays, records, and sums can support equality, similarity, and copying only when the underlying element, component, or variant types also support equality, similarity, and copying. To decide what an underlying type supports, we look at components of its value part.

S480c. ⟨*functions that build operations for equality, similarity, copying, and printing* S480b⟩+≡ ◁S480b S480d ▷

```
maybeXrComponent  : xrecord *      name -> value      option
maybeXrComponents : xrecord list * name -> value list option
```

```
fun maybeXrComponent (vp, f) =
  if xrHasComponent (vp, f) then SOME (xrComponent (vp, f)) else NONE

fun maybeXrComponents (vps, f) =
      optionList (map (fn vp => maybeXrComponent (vp, f)) vps)
```

Underlying operations for equivalence and copying are passed in. Because the type is immutable, `=` and `similar?` are the same.

S480d. ⟨*functions that build operations for equality, similarity, copying, and printing* S480b⟩+≡ ◁S480c S481a ▷

```
tau    : 'a map
mkEqv  : (value -> value -> bool) list -> ('a -> 'a -> bool)
mkCp   : (value -> value)         list -> ('a -> 'a)
argxrs : xrecord list
```

```
fun impPair (opname, mk) imps = Option.map (fn imps => (opname, mk imps)) imps

fun 'a mkImmutableEqualityOps tau { mkEqv, mkCp } argxrs =
  let fun cmp mk = efuncr (comparisonOf tau) o mk o map (pfunc (comparisonOf value))
      fun cpy mk = efuncr (copyOf       tau) o mk o map (pfunc (copyOf       value))
      fun complement eq a a' = not (eq a a')
      fun impsOf f = maybeXrComponents (argxrs, f)
  in  List.mapPartial id
      [ impPair ("=",  cmp mkEqv)                (impsOf "=")
      , impPair ("!=", cmp (complement o mkEqv))  (impsOf "=")
      , impPair ("similar?", cmp mkEqv)           (impsOf "similar?")
      , impPair ("copy",     cpy mkCp)            (impsOf "copy")
      ] : value ref env
  end
```

For a mutable type, we pass in an additional value, `identical`, which defines object identity.

S480e. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩≡ S483a ▷

S481a. ⟨*functions that build operations for equality, similarity, copying, and printing* S480b⟩+≡ ◁S480d S481b▷

```
fun mkMutableEqualityOps tau { identical, mkEqv, mkCp } argxrs =
  let fun cmp mk = efuncr (comparisonOf tau) o mk o map (pfunc (comparisonOf value))
      fun cpy mk = efuncr (copyOf       tau) o mk o map (pfunc (copyOf       value))
      fun impsOf f = maybeXrComponents (argxrs, f)
  in  List.mapPartial id
      [ SOME ("=",  efuncr (comparisonOf tau) (curry identical))
      , SOME ("!=", efuncr (comparisonOf tau) (curry (not o identical)))
      , impPair ("similar?",   cmp mkEqv) (impsOf "similar?")
      , impPair ("similar1?",  cmp mkEqv) (impsOf "=")
      , impPair ("copy",       cpy mkCp)  (impsOf "copy")
      , SOME ("copy1", efuncr (copyOf tau) (mkCp (map (fn _ => id) argxrs)))
      ]
  end
```

If each of the underlying types in `argxrs` has a `print` operation, then `mkPrintOps` delivers `print` and `println`.

S481b. ⟨*functions that build operations for equality, similarity, copying, and printing* S480b⟩+≡ ◁S481a

```
tau : 'a map
mkPrint : (value -> unit) list -> ('a -> unit)
```

```
fun 'a mkPrintOps tau mkPrint argxrs =
  let fun prn mk = efuncr (tau **-> unit) o mk o map (pfunc (value **-> unit))
      fun impsOf f = maybeXrComponents (argxrs, f)
      fun mkPrintln printers v = (mkPrint printers v; print "\n")
  in  List.mapPartial id
      [ impPair ("print",   prn mkPrint)   (impsOf "print")
      , impPair ("println", prn mkPrintln) (impsOf "print")
      ]
  end
```

T.3.4 Value parts of the built-in type constructors

Value part of type `int`

Most operations on integers can be implemented by predefined ML functions like `+`, `-`, and so on—but these functions have to be Curried. The exceptions are `power`, `from-to-by`, and `printu`.

S481c. ⟨*value parts of primitive clusters* `int`, `bool`, `sym`, *and* `null` S481c⟩≡ S482c▷

```
intXrecord : xrecord
```

```
val intXrecord =
  let ⟨definitions of functions power and from_to_by for int S482b⟩
  in  [ ("+",          efuncr (int **-> int **->> int) (curry op +))
      , ("-",          efuncr (int **-> int **->> int) (curry op -))
      , ("*",          efuncr (int **-> int **->> int) (curry op * ))
      , ("/",          efuncr (int **-> int **->> int) (curry op div))
      , ("negated",    efuncr (int **->> int)          ~)
      , ("mod",        efuncr (int **-> int **->> int) (curry op mod))
      , ("power",      efuncr (int **-> int **->> int) power)
      , ("max",        efuncr (int **-> int **->> int) (curry Int.max))
      , ("min",        efuncr (int **-> int **->> int) (curry Int.min))
      , ("abs",        efuncr (int **->> int)          Int.abs)
      , ("from-to-by", eiterr (int **->* int **->* int **->* iterator) from_to_by)
      , ("from-to",    eiterr (int **->* int **->* iterator) fromTo)
      , ("<",          efuncr (int **-> int **->> bool) (curry op <))
      , (">",          efuncr (int **-> int **->> bool) (curry op >))
      , ("<=",         efuncr (int **-> int **->> bool) (curry op <=))
      , (">=",         efuncr (int **-> int **->> bool) (curry op >=))
```

```
    , ("print",      efuncr (int **-> unit)           (print   o intString))
    , ("println",    efuncr (int **-> unit)           (println o intString))
    , ("printu",     efuncr (int **-> unit)           printUTF8)
    ]
    @ equalityOps int
  end
```

ML does not have power built in, so here it is. This version is deliberately inefficient; making power take logarithmic time is a homework problem.

S482a. ⟨*definitions of functions* power *and* from_to_by *for* int **[[prototype]]** S482a⟩≡

```
fun power base 0 = 1
  | power base n = base * power base (n - 1)
```

S482b. ⟨*definitions of functions* power *and* from_to_by *for* int S482b⟩≡ (S481c)

```
fun from_to_by low high by =
  if by < 0 then
    iterateLb (fn n => if n < high then NONE else SOME ([NUM n], n + by)) low
  else
    iterateLb (fn n => if n > high then NONE else SOME ([NUM n], n + by)) low
```

Value part of type bool

S482c. ⟨*value parts of primitive clusters* int, bool, sym, *and* null S481c⟩+≡ ◁S481c S482d▷

```
val boolXrecord =
  [ ("and",     efuncr (bool **-> bool **->> bool) (fn b => fn b' => b andalso b'))
  , ("or",      efuncr (bool **-> bool **->> bool) (fn b => fn b' => b orelse  b'))
  , ("not",     efuncr (bool **->> bool)           not)
  , ("print",   efuncr (bool **-> unit) (print   o valueString o BOOLV))
  , ("println", efuncr (bool **-> unit) (println o valueString o BOOLV))
  ]
  @ equalityOps bool
```

Value part of type null

S482d. ⟨*value parts of primitive clusters* int, bool, sym, *and* null S481c⟩+≡ ◁S482c S482e▷

```
val nullXrecord =
  [ ("print",   efuncr (nullmap **-> unit) (fn _ => print   "nil"))
  , ("println", efuncr (nullmap **-> unit) (fn _ => println "nil"))
  ]
  @ equalityOps nullmap
```

Value part of type sym

S482e. ⟨*value parts of primitive clusters* int, bool, sym, *and* null S481c⟩+≡ ◁S482d

```
val symXrecord =
  [ ("print",   efuncr (sym **-> unit) print)
  , ("println", efuncr (sym **-> unit) println)
  , ("hash",    efuncr (sym **->> int) fnvHash)
  ]
  @ equalityOps sym
```

Value parts of arrow types

We can't do much with routines, but they still get their primitives. A routine isn't equal to anything, even itself.

S483a. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S480e S483b▷

```
val arrowXrecord =
  let fun eq _ _ = false   (* can't compare possible LITERAL functions *)
  in  [ ("=",         efuncr (value **-> value **->> bool) eq)
      , ("!=",        efuncr (value **-> value **->> bool) (fn _ => fn _ => true))
      , ("similar?", efuncr (value **-> value **->> bool) eq)
      , ("copy",      efuncr (value **->> value) id)
      , ("print",     efuncr (value **-> unit) (fn _ => print "<routine>"))
      , ("println",   efuncr (value **-> unit) (fn _ => println "<routine>"))
      ]
  end
```

Value parts of array types

CLEAN ME UP LATER

S483b. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S483a S483c▷

```
fun fromTo low high : loop_body -> behavior =
  iterateLb (fn n => if n > high then NONE else SOME ([NUM n], n + 1)) low

val fromTo : int -> int -> loop_body -> behavior = fromTo

fun primStmt (f, es) = SETRESULTS ([], CALL (LITERAL (PRIMFUN f), ref NONE, es))

fun primLoopBody (f : value -> unit) =  (* YAGNI - as simple as possible *)
  let val rho = bind ("x", ref NONE, emptyEnv)
      val prim = fn [v] => (f v; [])
                  | _ => raise InternalError "wrong # of values from iterator loop"
  in  LB (([“x"], primStmt (prim, [VAR "x"])), rho, NONE)
  end
```

S483c. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S483b S485a▷

```
fun arrayXrecord (IMMUTABLE, elem) =
      let val array = iarray
          ⟨internal functions for immutable-array primitives S484a⟩
      in  [ ("new",     efuncr (result array) (Vector.fromList []))
          , ("empty?", efuncr (array **->> bool) (fn a => size a = 0))
          , ("at",      efuncr (array **-> int **->> value) (curry Vector.sub))
          , ("bottom", efuncr (array **->> value) (fn a => Vector.sub (a, 0)))
          , ("top",     efuncr (array **->> value) (fn a => Vector.sub (a, size a - 1)))
          , ("size",    efuncr (array **->> int) size)
          , ("elements", eiterr (array **->* iterator) vectorElements)
          , ("indices",  eiterr (array **->* iterator) vectorIndices)
          ]
          @ mkPrintOps array (aprint o single) [elem]
          @
          [ ("replace", efuncr (array **-> int **-> value **->> array) replace)
          , ("addh",    efuncr (array **-> value **->> array) addh)
          , ("addl",    efuncr (array **-> value **->> array) addl)
          , ("remh",    efuncr (array **->> array) remh)
          , ("reml",    efuncr (array **->> array) reml)
          , ("subseq", efuncr (array **-> int **-> int **->> array) subseq)
          , ("fill",    efuncr (int **-> value **->> array) fill)
          , ("e2a",     efuncr (value **->> array) (fn a => Vector.fromList [a]))
```

```
            , ("append", efuncr (array **-> array **->> array) append)
            , ("ia2ma",  efuncr (array **->> marray) ia2ma)
            , ("ma2ia",  efuncr (marray **->> array) ma2ia)
            ]
            @ mkImmutableEqualityOps array
                 { mkEqv = equal o single, mkCp = copy o single }
                 [elem]
        end
```

S484a. ⟨*internal functions for immutable-array primitives* S484a⟩≡ (S483c) S484b ▷

```
fun vectorIndices a = fromTo 1 (Vector.length a)
fun vectorElements a =
  let val size = Vector.length a
      fun next i =
        if i = size then NONE else SOME ([Vector.sub (a, i)], i + 1)
  in  iterateLb next 0
  end
```

S484b. ⟨*internal functions for immutable-array primitives* S484a⟩+≡ (S483c) ◁ S484a S484c ▷

```
val replace = curry3 Vector.update
val size = Vector.length
fun addh a v = Vector.concat [a, Vector.fromList [v]]
fun addl a v = Vector.concat [Vector.fromList [v], a]
fun remh a = Vector.tabulate (size a - 1, fn i => Vector.sub (a, i))
fun reml a = Vector.tabulate (size a - 1, fn i => Vector.sub (a, i+1))
```

S484c. ⟨*internal functions for immutable-array primitives* S484a⟩+≡ (S483c) ◁ S484b S484d ▷

```
fun subseq a start n =
  Vector.tabulate (n, fn i => Vector.sub (a, i + start))
fun fill n v = Vector.tabulate (n, fn _ => v)
(* XXX fill_copy XXX *)
fun append a a' = Vector.concat [a, a']
fun ma2ia a =
  let val bound = caBound a
  in  Vector.tabulate (caPop a, fn i => caAt (a, i + bound))
  end
fun ia2ma a = caNew (0, Vector.foldr op :: [] a)
```

S484d. ⟨*internal functions for immutable-array primitives* S484a⟩+≡ (S483c) ◁ S484c S484e ▷

```
fun aprint printElem a =
  ( print "(immutable array"
  ; Vector.app (fn v => (print " "; printElem v)) a
  ; print ")"
  )
```

S484e. ⟨*internal functions for immutable-array primitives* S484a⟩+≡ (S483c) ◁ S484d S484f ▷

```
fun equal elemEq a a' =
  Vector.length a = Vector.length a' andalso
  let fun cmp (x, y) = if elemEq x y then EQUAL else LESS
  in  Vector.collate cmp (a, a') = EQUAL
  end
```

S484f. ⟨*internal functions for immutable-array primitives* S484a⟩+≡ (S483c) ◁ S484e S484g ▷

```
fun copy elemCp a =
  Vector.tabulate (Vector.length a, fn i => elemCp (Vector.sub (a, i)))
```

S484g. ⟨*internal functions for immutable-array primitives* S484a⟩+≡ (S483c) ◁ S484f

```
fun single [imp] = imp
  | single _ = raise InternalError "wrong number of valpart args to array"

val eq = equal o single : (value -> value -> bool) list -> value vector -> value vector ->
```

S485a. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S483c S486b▷

```
| arrayXrecord (MUTABLE, elem) =
    let val array = marray
        ⟨internal functions for mutable-array primitives S485b⟩
    in  [ ("new",    ref (PRIMFUN (fn _ => [MARRAY (caNew (0, []))])))
        , ("empty?", efuncr (array **->> bool) (fn a => caPop a = 0))
        , ("at",     efuncr (array **-> int **->> value) (curry caAt))
        , ("bottom", efuncr (array **->> value) caBottom)
        , ("top",    efuncr (array **->> value) caTop)
        , ("size",   efuncr (array **->> int)   caPop)
        , ("elements", eiterr (array **->* iterator) elements)
        , ("indices",  eiterr (array **->* iterator) indices)
        ]
        @ mkPrintOps array (aprint o single) [elem]
        @
        [ ("create", efuncr (int **->> array) (fn n => caNew (n, [])))
        , ("low",    efuncr (array **->> int) caBound)
        , ("high",   efuncr (array **->> int) caHigh)
        , ("at-put", efuncr (array **-> int **-> value **-> unit) (curry3 caAtPut))
        , ("set-low", efuncr (array **-> int **-> unit) caSetLow)
        , ("fill",    efuncr (int **-> int **-> value **->> array) fill)
        , ("addh",   efuncr (array **-> value **-> unit) (curry caAddh))
        , ("addl",   efuncr (array **-> value **-> unit) (curry caAddl))
        , ("remh",   efuncr (array **->> value) caRemh)
        , ("reml",   efuncr (array **->> value) caReml)
        ]
        @
        mkMutableEqualityOps array
             { mkEqv = caSimilar o single, mkCp = caCopy o single, identical = caEq }
             [elem]
            (* not (curry op =): see http://mlton.org/PolymorphicEquality *)

        @ (if isbound ("copy", elem) then
             [ ("fill-copy", efuncr (int **-> int **-> value **->> array) fill_copy) ]
           else
             [ ])
        @
        [ ("copy1", efuncr (array **->> array) (caCopy id))
        ]

    end
```

§T.3 *Implementations of Molecule's primitive modules*

S485

S485b. ⟨*internal functions for mutable-array primitives* S485b⟩≡ (S485a) S485c▷

```
fun indices a = fromTo (caBound a) (caHigh a)
fun elements a =
  let val high = caHigh a
      fun next i = if i > high then NONE else SOME ([caAt (a, i)], i + 1)
  in  iterateLb next (caBound a)
  end
```

S485c. ⟨*internal functions for mutable-array primitives* S485b⟩+≡ (S485a) ◁S485b S485d▷

```
fun fill low n v = caNew (low, List.tabulate (n, fn _ => v))
fun fill_copy low n v =
  let val copy = pfunc (value **->> value) (!(find ("copy", elem)))
  in  caNew (low, List.tabulate (n, fn _ => copy v))
  end
```

S485d. ⟨*internal functions for mutable-array primitives* S485b⟩+≡ (S485a) ◁S485c S486a▷

```
fun aprint printElem a =
    ( app print ["(mutable array [at ", intString (caBound a), "]"]
```

```
      ; elements a (primLoopBody (fn v => (print " "; printElem v)))
      ; app print [")"]
      )
```

S486a. ⟨*internal functions for mutable-array primitives* S485b⟩+≡ (S485a) ◁S485d

```
fun single [imp] = imp
  | single _ = raise InternalError "wrong number of valpart args to array"
```


Value parts of record types

S486b. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S485a S486c▷

```
fun findField x r =
  find (x, r) handle NotFound _ => raise BugInTypeChecking "missing record field"
```

S486c. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S486b S487c▷

```
fun recordXrecord (IMMUTABLE, fields : (name * xrecord) list) =
      let val record = irecord
          ⟨internal functions for immutable-record primitives S486d⟩
      in
          [ ("ir2mr", efuncr (irecord **->> mrecord) (map (fn (x, v) => (x, ref v))))
          , ("mr2ir", efuncr (mrecord **->> irecord) (map (fn (x, r) => (x, !r))))
          ]
          @ fimps getOp
          @ fimps replaceOp
          @ mkImmutableEqualityOps record
              { mkEqv = eqRecords, mkCp = cpRecord }
              (map snd fields)
          @ mkPrintOps record mkPrint (map snd fields)
      end
```

S486d. ⟨*internal functions for immutable-record primitives* S486d⟩≡ (S486c) S486e▷

```
fun fimps f = map f fields
fun getOp (x, _) = ("get-" ^ x, efuncr (record **->> value) (findField x))
fun replaceField x r v =
  List.map (fn (x', v') => (x', if x = x' then v else v')) r
fun replaceOp (x, _) =
  ("replace-" ^ x, efuncr (record **-> value **->> record) (replaceField x))
```

S486e. ⟨*internal functions for immutable-record primitives* S486d⟩+≡ (S486c) ◁S486d S486f▷

```
fun checkFields r =
  if map fst r = map fst fields then
    ()
  else
    raise BugInTypeChecking ("field order in record value doesn't match " ^
                             "type (value " ^ spaceSep (map fst r) ^
                             ") vs (type " ^ spaceSep (map fst fields) ^ ")")
```

S486f. ⟨*internal functions for immutable-record primitives* S486d⟩+≡ (S486c) ◁S486e S487a▷

```
fun eqRecords argEqs r r' =
  ( checkFields r
  ; checkFields r'
  ; let fun all [] [] [] = true
          | all (eq::eqs) ((_, v)::fs) ((_, v')::fs') =
              eq v v' andalso all eqs fs fs'
          | all _ _ _ =
              raise BugInTypeChecking "wrong number of fields in record"
    in  all argEqs r r'
    end
  )
```

S487a. ⟨*internal functions for immutable-record primitives* S486d⟩+≡ (S486c) ◁ S486f S487b ▷

```
fun cpRecord argCps r =
  ( checkFields r
  ; let fun copy [] [] = []
            | copy (cp::cps) ((x, v)::fs) = (x, cp v) :: copy cps fs
            | copy _ _ =
                raise BugInTypeChecking "wrong number of fields in record"
    in  copy argCps r
    end
  )
```


S487b. ⟨*internal functions for immutable-record primitives* S486d⟩+≡ (S486c) ◁ S487a

```
fun mkPrint printers pairs =
  let fun printField (fp, (x, v)) =
        (print " ["; print x; print " "; fp v; print "]")
  in  ( print "(immutable record"
      ; ListPair.appEq printField (printers, pairs)
      ; print ")"
      )
  end
```

S487c. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁ S486c S488c ▷

```
  | recordXrecord (MUTABLE, fields) =
      let val record = mrecord
          ⟨internal functions for mutable-record primitives S487d⟩
      in
          [ ("mr_gets_mr", efuncr (mrecord **-> mrecord **-> unit) mr_gets_mr)
          , ("mr_gets_ir", efuncr (mrecord **-> irecord **-> unit) mr_gets_ir)
          ]
          @ fimps getOp
          @ fimps setOp
          @ mkMutableEqualityOps record
               { mkEqv = simRecords, mkCp = cpRecord, identical = op = }
               (map snd fields)
          @ mkPrintOps record mkPrint (map snd fields)
      end
```

S487d. ⟨*internal functions for mutable-record primitives* S487d⟩≡ (S487c) S487e ▷

```
fun fimps f = map f fields
fun setField x r v = findField x r := v
fun mr_gets_mr dst src =
      app (fn (x, cell) => cell := !(findField x src)) dst
fun mr_gets_ir dst src =
      app (fn (x, cell) => cell := findField x src) dst
fun getOp (x, _) = ("get-" ^ x, efuncr (record **->> value) (! o findField x))
fun setOp (x, _) =
  ("set-" ^ x, efuncr (record **-> value **-> unit) (setField x))
```

S487e. ⟨*internal functions for mutable-record primitives* S487d⟩+≡ (S487c) ◁ S487d S487f ▷

```
fun checkFields r =
  if map fst r = map fst fields then
    ()
  else
    raise BugInTypeChecking "field order in record value doesn't match type"
```

S487f. ⟨*internal functions for mutable-record primitives* S487d⟩+≡ (S487c) ◁ S487e S488a ▷

```
fun simRecords argEqs r r' =
  ( checkFields r
  ; checkFields r'
  ; let fun all [] [] [] = true
            | all (eq::eqs) ((_, vr)::fs) ((_, vr')::fs') =
```

```
              eq (!vr) (!vr') andalso all eqs fs fs'
          | all _ _ _ =
              raise BugInTypeChecking "wrong number of fields in record"
    in  all argEqs r r'
    end
  )
```

S488a. ⟨*internal functions for mutable-record primitives* S487d⟩+≡ (S487c) ◁S487f S488b▷


```
fun cpRecord argCps r =
  ( checkFields r
  ; let fun copy [] [] = []
          | copy (cp::cps) ((x, vr)::fs) = (x, ref (cp (!vr))) :: copy cps fs
          | copy _ _ =
              raise BugInTypeChecking "wrong number of fields in record"
    in  copy argCps r
    end
  )
```

S488b. ⟨*internal functions for mutable-record primitives* S487d⟩+≡ (S487c) ◁S488a

```
fun mkPrint printers pairs =
  let fun printField (fp, (x, ref v)) =
        (print " ["; print x; print " "; fp v; print "]")
  in  ( print "(mutable record"
      ; ListPair.appEq printField (printers, pairs)
      ; print ")"
      )
  end
```

Value parts of sum types

S488c. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S487c S488d▷

```
fun variantTagged variants x =
  find (x, variants)
  handle NotFound _ =>
    raise BugInTypeChecking ("unrecognized variant " ^ x)

fun printOneof (mutability, variants) (x, v) =
  ( app print ["(", mutabilityString mutability, " oneof [", x, " "]
  ; pfunc (value **-> unit) (xrComponent (variantTagged variants x, "print")) v
  ; print "])"
  )

fun eqOneof variantEqs (x, v) (x', v') = x = x' andalso (find (x, variantEqs) v v')
fun cpOneof variantCps (x, v) = (x, find (x, variantCps) v)
```

S488d. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S488c S489▷

```
fun oneofXrecord (IMMUTABLE, variants : (name * xrecord) list) =
      let val oneof = ioneof
          val vt = variantTagged variants
          fun vimps f = map f variants
          fun makeOp (x, _) =
            ("make-" ^ x,      efuncr (value **->> oneof) (fn v => (x, v)))
          fun isOp   (x, _) =
            ("is-" ^ x ^ "?", efuncr (oneof **->> bool)  (fn (x', _) => x = x'))
          fun valueOp (x, _) =
            ("value-" ^ x, efuncr (oneof **->> value)
                             (fn (x', v) => if x = x' then
                                                v
```

```
                                     else
                                       raise RuntimeError ("applied value-" ^ x ^
                                                           ", but tag is " ^ x')))

        fun tag functions = ListPair.zip (map fst variants, functions)

        fun mkPrint _ = printOneof (IMMUTABLE, variants)
    in
        List.concat (map vimps [makeOp, isOp, valueOp])
        @
        [ ("io2mo", efuncr (ioneof **->> moneof) ref)
        , ("mo2io", efuncr (moneof **->> ioneof) !)
        ]
        @ mkImmutableEqualityOps oneof { mkEqv = eqOneof o tag, mkCp = cpOneof o tag }
              (map snd variants)
        @ mkPrintOps oneof mkPrint (map snd variants)
    end
```


S489. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S488d S490a▷

```
  | oneofXrecord (MUTABLE, variants) =
    let val oneof = moneof
        val vt = variantTagged variants
        fun vimps f = map f variants
        fun makeOp (x, _) =
          ("make-" ^ x,      efuncr (value **->> oneof) (fn v => ref (x, v)))
        fun changeOp (x, _) =
          ("change-" ^ x, efuncr (oneof **-> value **-> unit)
                                 (fn cell => fn v => cell := (x, v)))
        fun isOp    (x, _) =
          ("is-" ^ x ^ "?", efuncr (oneof **->> bool)  (fn (ref (x', _)) => x = x'))
        fun valueOp (x, _) =
          ("value-" ^ x, efuncr (oneof **->> value)
                             (fn (ref (x', v)) =>
                                if x = x' then
                                  v
                                else
                                  raise RuntimeError ("applied value-" ^ x ^
                                                      ", but tag is " ^ x')))

        fun mkEqv variants one one' = eqOneof variants (!one) (!one')
        fun mkCp  variants one = ref (cpOneof variants (!one))
        fun tag functions = ListPair.zip (map fst variants, functions)

        fun mkPrint _ = printOneof (IMMUTABLE, variants) o !
    in
        List.concat (map vimps [makeOp, isOp, valueOp, changeOp])
        @
        [ ("mo_gets_mo", efuncr (oneof **-> oneof **-> unit)
                                                     (fn c => fn c' => c := !c'))
        , ("mo_gets_io", efuncr (oneof **-> ioneof **-> unit)
                                                     (fn c => fn pair => c := pair))
        ]
        @ mkMutableEqualityOps oneof
              { identical = op =, mkEqv = mkEqv o tag, mkCp = mkCp o tag }
              (map snd variants)
        @ mkPrintOps oneof mkPrint (map snd variants)
    end
```

This function tests to make sure an export record is consistent with its type.

S490a. ⟨*evaluation of the value parts of array, record, sum, and arrow types* S480e⟩+≡ ◁S489

```
fun exportSanityCheck (what, exports, xr) =
  let fun checkType (x, tau) =
        if isbound (x, xr) then ()
        else
          raise InternalError (what ^ " claims to export " ^ x ^ " : " ^
                                  typeString tau ^ ", but it's not in the export record")
      fun checkValue (x, ref v) =
        if isbound (x, exports) then ()
        else
          raise InternalError (what ^ " exports value " ^ x ^ " = " ^
                                  valueString v ^ ", but it's not in the type")
  in  ( app checkType exports
      ; app checkValue xr
      )
  end
```

T.3.5 *The initial basis*

S490b. ⟨*implementations of Molecule primitives and definition of* `initialBasis` S490b⟩≡ (S501a)

```
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

val modules =
  [ ("Int",  intmod,  MODVAL intmodenv)
  , ("Bool", boolmod, MODVAL boolmodenv)
  , ("Unit", unitmod, MODVAL unitmodenv)
  , ("Sym",  symmod,  MODVAL symmodenv)
  , (arraymodname,  arraymod,
     CLOSURE ((["Elem"], MODEXP (map (fn (x, f, _) => (x, LITERAL (PRIMITIVE f))) arrayPrim
              emptyEnv))
  , ("UnsafeArray",  uarraymod,
     CLOSURE ((["Elem"], MODEXP (map (fn (x, f, _) => (x, LITERAL (PRIMITIVE f))) uarrayPri
              emptyEnv))
  , ("ArrayCore",  arraymod,
     CLOSURE ((["Elem"], MODEXP (map (fn (x, f, _) => (x, LITERAL (PRIMITIVE f))) arrayPrim
              emptyEnv))

  , ("#t", ENVVAL booltype, CONVAL (PNAME "#t", []))
  , ("#f", ENVVAL booltype, CONVAL (PNAME "#f", []))
  ]

fun addmod ((x, dbl, v), (Gamma, rho)) =
  (bind (x, dbl, Gamma), bind (x, ref v, rho))

val initialRho = bind (overloadTable, ref (ARRAY emptyOverloadTable), emptyEnv)

val initialBasis = foldl addmod (emptyEnv, initialRho) modules : basis

val initialBasis =
  let val predefinedTypes = ⟨predefined Molecule types, functions, and modules, as strings generated automa
      val xdefs = stringsxdefs ("built-in types", predefinedTypes)
  in  readEvalPrintWith predefinedFunctionError (xdefs, initialBasis, noninteractive)
```

```
  end

val options = case OS.Process.getEnv "BPCOPTIONS" of SOME s => ":" ^ s ^ ":" | NONE => ""
val () =
  if String.isSubstring ":basis:" options then
    let fun show (x, c) = app print [whatdec c, " ", x, "\n"]
    in  app show (fst initialBasis)
    end
  else
    ()
```


S491a. ⟨*predefined Molecule types, functions, and modules* S475a⟩+≡ ◁ S475b S494a ▷

```
(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.3.6 The initial basis

S491b. ⟨*primitive modules and types used to type literal expressions* S491b⟩≡ (S500c)

```
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"))

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"


fun addValWith f ((x, v, ty), rho) = bind (x, f v, rho)
fun decval (x, v, ty) = (x, ENVVAL ty)
fun compval (x, v, ty) = (x, COMPVAL ty)


⟨shared utility functions for building primitives in languages with type checking S389d⟩
⟨primitives [[mcl]] S492a⟩

val unitval =
  ("unit", CONVAL (PNAME "unit", []), TYNAME (PDOT (PNAME unitmodident, "t")))

local
  fun module id primvals : binding =
    ENVMOD (MTEXPORTS (("t", COMPABSTY (PDOT (PNAME id, "t"))) :: map compval pr
            PNAME id)
in
```

```
    val intmod  = module intmodident intPrims
    val symmod  = module symmodident symPrims
    val boolmod = module boolmodident boolPrims
    val unitmod = module unitmodident [unitval]
    val arraymod  = ENVMOD (arraymodtype, PNAME arraymodident)
    val uarraymod  = ENVMOD (uarraymodtype, PNAME uarraymodident)
  end
```

S492a. ⟨*primitives* **[[mcl]]** S492a⟩≡ (S491b) S492b ▷

```
  fun eqPrintPrims tau strip =
    let val comptype = FUNTY ([tau, tau], booltype)
        fun comparison f = binaryOp (embedBool o (fn (x, y) => f (strip x, strip y)))
    in  ("similar?",  comparison op =,  comptype) ::
        ("dissimilar?",  comparison op =,  comptype) ::
        ("=",  comparison op =,  comptype) ::
        ("!=", comparison op <>, comptype) ::
        ("print", unaryOp (fn x => (print (valueString x);unitVal)), FUNTY ([tau], unittype))
        ("println", unaryOp (fn x => (println (valueString x);unitVal)), FUNTY ([tau], unitty
        []
    end

  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")
```

S492b. ⟨*primitives* **[[mcl]]** S492a⟩+≡ (S491b) ◁ S492a S493 ▷

```
  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 asInt (NUM n) = n
    | asInt v = raise BugInTypeChecking ("expected a number; got " ^ valueString v)

  val arithtype = FUNTY ([inttype, inttype], inttype)
  val comptype  = FUNTY ([inttype, inttype], booltype)

  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)


  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], uni
  eqPrintPrims inttype (fn NUM n => n | _ => raise BugInTypeChecking "comparing non-numbers"))
```

S493. ⟨*primitives* **[[mcl]]** S492a⟩+≡ (S491b) ◁S492b S534d▷

```
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"))


  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); unitVa
                | _ => raise BugInTypeChecking "number or types of args to Array
            FUNTY ([arraytype, inttype, elemtype], unittype)) ::
    []

  val arraymodtype : modty =
    MTARROW ([(arrayarg, MTEXPORTS [("t", COMPABSTY (PDOT (argpath, "t")))]   : m
             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, CONVAL (PNAME
                       | _ => raise BugInTypeChecking "array size not a number")
            FUNTY ([inttype], arraytype)) ::
    []

  val uarraymodtype : modty =
    MTARROW ([(arrayarg, MTEXPORTS [("t", COMPABSTY (PDOT (argpath, "t")))]   : m
```

`[])`

arithOp	S389e
ARRAY	S499d
arraymodident	S491b
binaryOp	S389d
booltype	S491b
BugInTypeChecking	S237b
COMPABSTY	S456b
COMPMANTY	S456b
compval	S491b
CONVAL	S499d
embedBool	S433d
FUNTY	S456a
genmodident	S494c
id	S263d
inttype	S491b
type modty	S456b
MTARROW	S456b
MTEXPORTS	S456b
NUM	S499d
PAPPLY	S455
PDOT	S455
PNAME	S455
println	S238a
printUTF8	S239b
RuntimeError	S366c
SYM	S499d
symtype	S491b
TYNAME	S456a
unaryOp	S389d
unittype	S491b
unitVal	S500b
valueString	S507a

```
              MTEXPORTS (("t", COMPABSTY (PDOT (resultpath, "t"))) ::
                         map compval uarrayPrims) : modty)
end
```

S494a. ⟨*predefined Molecule types, functions, and modules* S475a⟩+≡ ◁S491a S494b▷

```
(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))
```


S494b. ⟨*predefined Molecule types, functions, and modules* S475a⟩+≡ ◁S494a

```
(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.4 Refugees from the chapter (type checking)

T.4.1 Path and type basics

S494c. ⟨*definition of function* `genmodident` S494c⟩≡ (S455)

```
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 (* XXX fix this later *)
        val _ = timesDefined := bind (name, n + 1, !timesDefined)
    in  MODCON { printName = name, serial = n }
    end
end
```

S494d. ⟨*paths for Molecule* S455⟩+≡ (S500b) ◁S455

```
fun plast (PDOT (_, x)) = x
  | plast (PNAME (_, x)) = x
  | plast (PAPPLY _) = "??last??"
```

S494e. ⟨*type equality for Molecule* S494e⟩≡ (S500c)

```
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)
```


T.4.2 Substitutions (boring)

S495a. ⟨*substitutions for Molecule* S495a⟩≡ (S500c 501a) S495b ▷

```
type rootsubst = (modident * path) list
val idsubst = []
```

```
type rootsubst
idsubst : rootsubst
```

S495b. ⟨*substitutions for Molecule* S495a⟩+≡ (S500c 501a) ◁ S495a S495c ▷

```
infix 7 |-->
fun id |--> p = [(id, p)]
```

```
|--> : modident * path -> rootsubst
```

S495c. ⟨*substitutions for Molecule* S495a⟩+≡ (S500c 501a) ◁ S495b S495d ▷

```
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
```

```
type tysubst
associatedWith : path * tysubst -> ty option
hasKey : tysubst -> path -> bool
```

S495d. ⟨*substitutions for Molecule* S495a⟩+≡ (S500c 501a) ◁ S495c S495e ▷

```
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
```

```
pathsubstRoot : rootsubst -> path -> path
```

S495e. ⟨*substitutions for Molecule* S495a⟩+≡ (S500c 501a) ◁ S495d S495f ▷

```
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
```

S495f. ⟨*substitutions for Molecule* S495a⟩+≡ (S500c 501a) ◁ S495e S496a ▷

```
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
```

```
dom : rootsubst -> modident set
compose : rootsubst * rootsubst -> rootsubst
```

ANYTYPE	S456a
bind	312b
emptyEnv	311a
type env	310b
find	311b
FUNTY	S456a
MODCON	S455
type modident	S455
NotFound	311b
PAPPLY	S455
type path	S455
PDOT	S455
PNAME	S455
type ty	S456a
TYNAME	S456a
union	S240b

S496a. ⟨*substitutions for Molecule* S495a⟩+≡ (S500c 501a) ◁S495f S496b▷

```
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
```

Supporting code for Molecule

S496

S496b. ⟨*substitutions for Molecule* S495a⟩+≡ (S500c 501a) ◁S496a S496c▷

```
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
```

S496c. ⟨*substitutions for Molecule* S495a⟩+≡ (S500c 501a) ◁S496b

```
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.4.3 Realization

This general-purpose code ought to go elsewhere.

S496d. ⟨*utilities for module-type realization* S496d⟩≡ (S500c) S497a▷

```
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
```

S497a. ⟨*utilities for module-type realization* S496d⟩+≡ (S500c) ◁ S496d

```
fun emptyExports (MTEXPORTS []) = true
  | emptyExports _ = false
```


Restores the invariant at need.

S497b. ⟨*module-type realization* S458c⟩+≡ (S500c) ◁ S459b

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.4.4 Instantiation

S497c. ⟨*instantiated exporting module* `fpx` S497c⟩≡ (S461a)

```
raise TypeError ("module " ^ pathexString fpx ^ " is an exporting module, and only " ^
                 " a generic module can be instantiated")
```

S497d. ⟨*can't pass* `actroot` *as* `formalid` *to* `fpx` S497d⟩≡ (S461a)

```
raise TypeError ("module " ^ pathString actroot ^ " cannot be used as argument '
                 modidentString formalid ^ " to generic module " ^ pathexString
                 ": " ^ msg)
```

S497e. ⟨*wrong number of arguments to* `fpx` S497e⟩≡ (S461a)

```
raise TypeError ("generic module " ^ pathexString fpx ^ " is expecting " ^
                 countString formals "parameter" ^ ", but got " ^
                 countString actuals "actual parameter")
```

T.4.5 Translation/elaboration of syntax into types

We translate paths, types, declarations, and module types.

S497f. ⟨*translation of Molecule type syntax into types* S497f⟩≡ (S500c 501a) S498a ▷

txpath : pathex * binding env -> path

```
fun txpath (px, Gamma) =
  let fun tx (PAPPLY (f, args)) = PAPPLY (tx f, map tx args)
        | tx (PDOT (p, x)) = PDOT (tx p, x)
        | tx (PNAME (loc, m)) =
            let fun bad aThing =
                  raise TypeError ("I was expecting " ^ m ^ " to refer to a modu
                                   "but at " ^ srclocString loc ^ ", it's " ^ a
            in  case find (m, Gamma)
                  of ENVMODTY _ => bad "a module type"
                   | ENVMOD (mt, p) => p
                   | c => bad (whatdec c)
            end
  in  tx px
  end
val elabpath = txpath
```

S498a. ⟨*translation of Molecule type syntax into types* S497f⟩+≡ (S500c 501a) ◁S497f S498b▷

`elabty : tyex * binding env -> ty`

```
fun elabty (t, Gamma) =
  let fun tx (TYNAME px) =
            (case pathfind (px, Gamma)
               of ENVMANTY tau => tau
                | dec => raise TypeError ("I was expecting a type, but " ^
                                          pathexString px ^ " is " ^ whatdec dec))
        | tx (FUNTY (args, res)) = FUNTY (map tx args, tx res)
        | tx ANYTYPE = ANYTYPE
  in  tx t
  end
```

S498b. ⟨*translation of Molecule type syntax into types* S497f⟩+≡ (S500c 501a) ◁S498a S498c▷

`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")
```

S498c. ⟨*translation of Molecule type syntax into types* S497f⟩+≡ (S500c 501a) ◁S498b S499b▷

`elabmt : modtyx rooted * binding env -> modty`

```
fun elabmt ((mtx : modtyx, path), Gamma) =
  let fun tx (MTNAMEDX t) = mtsubstRoot (MODTYPLACEHOLDER t |--> path) (findModty (t, Gamma
        | tx (MTEXPORTSX exports) =
             let val (this', _) = foldl (leftLocated export) ([], Gamma) exports
             in  MTEXPORTS (rev this')
             end
        | tx (MTALLOFX mts) = allofAt (map (located tx) mts, path)
        | tx (MTARROWX (args, body)) =
            let val resultName = PNAME (MODTYPLACEHOLDER "functor result")
                fun txArrow ([], (loc, body), Gamma : binding env, idents') =
                      let val resultName = PAPPLY (path, reverse idents')
                      in
                      ([], atLoc loc elabmt ((body, resultName), Gamma))
                      end
                  | txArrow (((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 S499a⟩
                          val Gamma' = bind (m, ENVMOD (modty, PNAME modid), Gamma)
                             (* XXX check 1st arg to ENVMOD *)
                          val (rest', body') = txArrow (rest, body, Gamma', PNAME modid ::
                      in  ((modid, modty) :: rest', body')
                      end
            in  MTARROW (txArrow (args, body, Gamma, []))
            end

      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 = txComp ((ctx, PDOT (path, x)), Gamma)
              in  ((x, c) :: theseDecls, bind (x, asBinding (c, path), Gamma))
              end
  in  tx mtx
```

```
  end
```

S499a. ⟨*if* `modty` *is generic, bleat about* `m` S499a⟩≡ (S498c)

```
case modty
  of MTARROW _ =>
    raise TypeError ("module parameter " ^ m ^ " is generic, but a generic " ^
                     "module may not take another generic module as a parameter")
   | _ => ()
```

S499b. ⟨*translation of Molecule type syntax into types* S497f⟩+≡ (S500c 501a) ◁S498c

```
txDecl     : decl rooted * binding env -> binding
txComp     : decl rooted * binding env -> component
```

```
and txComp ((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))
                                (* XXX is path really OK here??? *)
         | DECMODTY mt =>
             raise TypeError ("module type " ^ pathString path ^ " may not be a
  end
and txDecl ((comp : decl, path), Gamma : binding env) : binding =
  let fun ty t = elabty (t, Gamma)
  in  case comp
        of DECVAL tau   => ENVVAL (ty tau)
         | DECABSTY     => ENVMANTY (TYNAME path)
         | DECMANTY t   => ENVMANTY (ty t)
         | DECMOD mt    => ENVMOD (elabmt ((mt, path), Gamma), path)
                                (* XXX is path really OK here??? *)
         | DECMODTY mt => ENVMODTY (elabmt ((mt, path), Gamma))
  end
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
```

S499c. ⟨*tried to select* `path.x` *but* `path` *is a* `dec` S499c⟩≡ (S460)

```
raise TypeError ("Tried to select " ^ pathexString (PDOT (path, x)) ^ ", but " ^
                 pathexString path ^ " is " ^ whatdec dec ^ ", which does not "
                 " have components")
```

T.4.6 *Exp and value representations*

S499d. ⟨*definitions of* `exp` *and* `value` *for Molecule* S462a⟩+≡ (S500b) ◁S462a

```
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
```

S500a. ⟨*translation of definition list of* `MODEXP` S500a⟩≡
```
fun modexp defs =
  let fun bindings [] = []
        | bindings (d :: ds) =
```

The representations defined above are combined with representations from other chapters as follows:

S500b. ⟨*abstract syntax and values for Molecule* S500b⟩≡ (S501a)
```
⟨paths for Molecule S455⟩
⟨definition of ty for Molecule S456a⟩
⟨definition of modty for Molecule S456b⟩
type vcon = name path'
datatype pat = WILDCARD
             | PVAR     of name
             | CONPAT   of vcon * pat list
⟨definitions of exp and value for Molecule S462a⟩
val unitVal = SYM "unit"  (* XXX placeholder *)
⟨definition of def for Molecule S462b⟩
(*<definition of [[implicit_data_def]] for \mcl>*)
⟨definition of unit_test for explicitly typed languages generated automatically⟩
  | CHECK_MTYPE of pathex * modtyx
⟨definition of xdef (shared) S365b⟩
val BugInTypeInference = BugInTypeChecking (* to make \uml utils work *)
⟨definition of valueString for Molecule S507a⟩
⟨definition of patString for μML and μHaskell generated automatically⟩
⟨definition of typeString for Molecule types S531b⟩
⟨definition of expString for Molecule S532d⟩
⟨utility functions on μML values generated automatically⟩
```

T.4.7 Wrapup

S500c. ⟨*type checking for Molecule* S500c⟩≡ (S501a)
```
⟨context for a Molecule definition S465b⟩
⟨type equality for Molecule S494e⟩
⟨substitutions for Molecule S495a⟩
⟨type components of module types S457a⟩
⟨utilities for module-type realization S496d⟩
⟨module-type realization S458c⟩
⟨invariants of Molecule S457b⟩
⟨implements relation, based on subtype of two module types S457c⟩
⟨path-expression lookup S460⟩
⟨translation of Molecule type syntax into types S497f⟩
⟨primitive modules and types used to type literal expressions S491b⟩
⟨utility functions on Molecule types S463a⟩
⟨typeof a Molecule expression generated automatically⟩
⟨principal type of a module S465a⟩
⟨elaboration and evaluation of data definitions for Molecule S469b⟩
⟨elaborate a Molecule definition S466a⟩
```

S500d. ⟨*support for operator overloading in Molecule* S500d⟩≡ (S501a)
```
val notOverloadedIndex = ~1
val overloadTable = "overloaded operators"  (* name cannot appear in source code *)
val emptyOverloadTable = Array.tabulate (10, fn _ => SYM "<empty entry in overload table>")
fun overloadCell rho =
  find (overloadTable, rho) handle NotFound _ => raise InternalError "missing overload tabl
fun overloadedAt (rho, i) =
  case overloadCell rho
    of ref (ARRAY a) => Array.sub (a, i)
```

```
      | _ => raise InternalError "representation of overload table"
  local
    val next = ref 0
  in
    fun nextOverloadedIndex () = !next before next := !next + 1
  end

  fun overloadedPut (i, v, rho) =
    let val cell = overloadCell rho
        val a  = case cell of ref (ARRAY a) => a | _ => raise InternalError "rep of overload table"
        val a' = if i >= Array.length a then
                   let val n = 2 * Array.length a
                       val a' = Array.tabulate (n, fn j => if j < n then Array.sub (a, j) else v)
                       val _ = cell := ARRAY a'
                   in  a'
                   end
                 else
                   a
    in  Array.update (a', i, v)
    end
```

S501a. ⟨*mcl.sml* S501a⟩≡

```
exception Unimp of string
fun unimp s = raise Unimp s
```

⟨*exceptions used in languages with type checking* S237b⟩
⟨*shared: names, environments, strings, errors, printing, interaction, streams, & initialization* S237a⟩

⟨*abstract syntax and values for Molecule* S500b⟩
⟨*support for operator overloading in Molecule* S500d⟩
⟨*lexical analysis and parsing for Molecule, providing* `filexdefs` *and* `stringsxdefs` S517c⟩

```
(*<\mcl's overloaded operators>*)
```

⟨*environments for Molecule's defined names* S507c⟩

⟨*type checking for Molecule* S500c⟩

⟨*substitutions for Molecule* S495a⟩

⟨*translation of Molecule type syntax into types* S497f⟩
⟨*type checking for Molecule* S500c⟩
⟨*evaluation, testing, and the read-eval-print loop for Molecule* S501b⟩

⟨*implementations of Molecule primitives and definition of* `initialBasis` S490b⟩
⟨*function* `runAs`*, which evaluates standard input given* `initialBasis` S372c⟩
⟨*code that looks at command-line arguments and calls* `runAs` *to run the interpreter* S372d⟩

T.5 Evaluation

The components of the evaluator and read-eval-print loop are organized as follows:

S501b. ⟨*evaluation, testing, and the read-eval-print loop for Molecule* S501b⟩≡ (S501a)

⟨*definition of* `namedValueString` *for functional bridge languages* S505a⟩

```
fun basename (PDOT (_, x)) = PNAME x
  | basename (PNAME x) = PNAME x
  | basename (instance as PAPPLY _) = instance
```

⟨*definitions of* `matchRef` *and* `Doesn'tMatch` generated automatically⟩
⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S502a⟩

⟨*definitions of* `basis` *and* `processDef` *for Molecule* S471a⟩

⟨*shared definition of* `withHandlers` S371a⟩
⟨*shared unit-testing utilities* S246d⟩
⟨*definition of* `testIsGood` *for Molecule* S526d⟩
```
fun assertPtype (x, t, basis) = unimp "assertPtype"
```

⟨*shared definition of* `processTests` S247b⟩
⟨*shared read-eval-print loop and* `processPredefined` S369a⟩

T.5.1 Evaluating paths

S502a. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S502a⟩≡ (S501b) S502b ▷
```
val nullsrc : srcloc = ("translated name in LETRECX", ~1)

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.5.2 Evaluating expressions

The implementation of the evaluator is almost identical to the implementation in Chapter 5. There are only two significant differences: we have to deal with the mismatch in representations between the abstract syntax `LAMBDA` and the value `CLOSURE`, and we have to write cases for the `TYAPPLY` and `TYLAMBDA` expressions. 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, we raise the exception `BugInTypeChecking`, not `RuntimeError`.

S502b. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S502a⟩+≡ (S501b) ◁ S502a S504d ▷
```
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 S502c⟩
        | ev (EXP_AT (loc, e)) = atLoc loc ev e
  in  ev e
  end
```

Code for variables is just as in Chapter 5.

S502c. ⟨*more alternatives for* `ev` *for Molecule* S502c⟩≡ (S502b) S503a ▷
```
| ev (VAR p) = evalpath (p, rho)
| ev (SET (n, e)) =
    let val v = ev e
```

```
in  find (n, rho) := v;
    unitVal
end
```

S503a. ⟨*more alternatives for* ev *for Molecule* S502c⟩+≡ (S502b) ◁ S502c S503b ▷

```
| ev (VCONX c) = evalpath (addloc ("bogus", ~33) c, rho)
| ev (CASE (LITERAL v, (p, e) :: choices)) =
    (let val rho' = matchRef (p, v)
     in  eval (e, extend (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.

S503b. ⟨*more alternatives for* ev *for Molecule* S502c⟩+≡ (S502b) ◁ S503a S503c ▷

```
| 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.

S503c. ⟨*more alternatives for* ev *for Molecule* S502c⟩+≡ (S502b) ◁ S503b S503d ▷

```
| 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, we raise BugInTypeChecking, not RuntimeError, because the type checker should reject any program that could apply a non-function.

S503d. ⟨*more alternatives for* ev *for Molecule* S502c⟩+≡ (S502b) ◁ S503c S503e ▷

```
| 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 inde
               | _ => raise BugInTypeChecking "overloaded name is not array"
   in  case fv
         of PRIMITIVE prim => prim (map ev args)
          | CLOSURE clo => ⟨apply closure clo to args 317b⟩
          | v => raise BugInTypeChecking "applied non-function"
   end
```

Code for the LETX family is as in Chapter 5.

S503e. ⟨*more alternatives for* ev *for Molecule* S502c⟩+≡ (S502b) ◁ S503d S504a ▷

```
| 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
```

S504a. ⟨*more alternatives for* ev *for Molecule* S502c⟩+≡ (S502b) ◁S503e S504b▷


```
    | ev (LETRECX (bs, body)) =
        let val (lhss, values) = ListPair.unzip bs
            val names = map fst lhss
            val _ = errorIfDups ("bound name", names, "letrec")
            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
```

S504b. ⟨*more alternatives for* ev *for Molecule* S502c⟩+≡ (S502b) ◁S504a S504c▷

```
    | 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
```

S504c. ⟨*more alternatives for* ev *for Molecule* S502c⟩+≡ (S502b) ◁S504b

```
    | ev (ERRORX es) =
        raise RuntimeError (spaceSep (map (valueString o ev) es))
```

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.

S504d. ⟨*definitions of* `eval` *and* `evaldef` *for Molecule* S502a⟩+≡ (S501b) ◁S502b S505b▷

```
                 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 (QNAME _, 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.

The string returned by evaldef is the value, unless the value is a named procedure, in which case it is the name.

S505a. ⟨*definition of* namedValueString *for functional bridge languages* S505a⟩≡ (S501b)

```
fun namedValueString x v =          namedValueString : name -> value -> string
  case v of CLOSURE ((_, MODEXP _), _) => "generic module " ^ x
          | CLOSURE _ => x
          | PRIMITIVE _ => x
          | MODVAL _ => "module " ^ x
          | _ => valueString v
```

XXX *I probably should evaluate a definition by using* defexps *and* eval.

S505b. ⟨*definitions of* eval *and* evaldef *for Molecule* S502a⟩+≡ (S501b) ◁S504d S505c▷

```
  | defbindings (TYPE _, _) =
      []
  | 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
  | 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)))]
  | defbindings (MODULETYPE (a, _), rho) =
      []
```

S505c. ⟨*definitions of* eval *and* evaldef *for Molecule* S502a⟩+≡ (S501b) ◁S505b S505d▷

```
  | 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
```

S505d. ⟨*definitions of* eval *and* evaldef *for Molecule* S502a⟩+≡ (S501b) ◁S505c S505e▷

```
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
```

S505e. ⟨*definitions of* eval *and* evaldef *for Molecule* S502a⟩+≡ (S501b) ◁S505d S506a▷

```
and defexps (VAL (x, e)) = [(x, e)]
  | defexps (VALREC (x, tau, e)) = [(x, LETRECX ([((x, tau), e)], VAR (PNAME (nu
  | defexps (EXP e) =  [("it", e)]
  | defexps (QNAME _) = []
  | defexps (DEFINE (f, tau, lambda)) = defexps (VALREC (f, tau, LAMBDA lambda))
```

Supporting code for Molecule

S506

```
  | defexps (TYPE _) = []
  | defexps (DATA (t, typed_vcons)) =
      let fun isfuntype (FUNTY _)          = true
            | isfuntype _                  = false
          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"
```

S506a. ⟨*definitions of* eval *and* evaldef *for Molecule* S502a⟩+≡ (S501b) ◁ S505e S506b ▷

```
and modexp (MPATH px)             = VAR px
  | modexp (MPATHSEALED (_, px)) = VAR px
  | modexp (MSEALED (_, defs))    = MODEXP ((List.concat o map (located defexps)) defs)
  | modexp (MUNSEALED defs)       = MODEXP ((List.concat o map (located defexps)) defs)
```

S506b. ⟨*definitions of* eval *and* evaldef *for Molecule* S502a⟩+≡ (S501b) ◁ S506a S506c ▷

```
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)))
                 (* XXX type checker should ensure there are no duplicates here *)
```

S506c. ⟨*definitions of* eval *and* evaldef *for Molecule* S502a⟩+≡ (S501b) ◁ S506b S506d ▷

```
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
```

S506d. ⟨*definitions of* eval *and* evaldef *for Molecule* S502a⟩+≡ (S501b) ◁ S506c

```
and evaldef (d, rho) =
```

```
evaldef : baredef * value ref env -> value ref env * value list
```

```
    let fun single [(_, loc)] = ! loc
          | single _          = raise InternalError "wrong number of bindings from def"
      val bindings = defbindings (d, rho)

      fun string (VAL (x, e))          = namedValueString x (single bindings)
        | string (VALREC (x, tau, e)) = namedValueString x (single bindings)
        | string (EXP _)               = valueString (single bindings)
        | string (QNAME px)            = raise InternalError "NAME reached evaldef"
        | string (DEFINE (f, _, _))    = namedValueString f (single bindings)
        | string (TYPE (t, tau))       = "type " ^ t
        | string (DATA _) = unimp "DATA definitions"
        | string (GMODULE (f, _, _))= namedValueString f (single bindings)
        | string (MODULE (x, m))       = namedValueString x (single bindings)
        | string (MODULETYPE (a, _))   = "module type " ^ a
        | string (OVERLOAD ps)         = "overloaded names " ^ separate("", " ") (map plast
```

```
    val rho' = foldl (fn ((x, loc), rho) => bind (x, loc, rho)) rho bindings
  in  (rho', map (! o snd) bindings)   (* 2nd component was (string d) *)
  end
```

Practically duplicates μML. Can we share code?

S507a. ⟨*definition of* `valueString` *for Molecule* S507a⟩≡ (S500b) S507b ▷

```
fun vconString (PNAME c) = c
  | vconString (PDOT (m, c)) = vconString m ^ "." ^ c
  | vconString (PAPPLY _) = "can't happen! (vcon PAPPLY)"

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.toS
  | valueString (SYM v      )  = v
  | valueString (CLOSURE   _)  = "<function>"
  | valueString (PRIMITIVE _)  = "<function>"
  | valueString (MODVAL _)     = "<module>"
  | valueString (ARRAY a)      =
      "[" ^ spaceSep (map valueString (Array.foldr op :: [] a)) ^ "]"
```

S507b. ⟨*definition of* `valueString` *for Molecule* S507a⟩+≡ (S500b) ◁ S507a

```
and consString (v, vs) =
      let fun tail (CONVAL (PNAME "cons", [ref v, ref vs])) = " " ^ valueString
            | tail (CONVAL (PNAME "'()", []))       = ")"
            | tail _ =
                raise BugInTypeChecking
                  "bad list constructor (or cons/'() redefined)"
      in  "(" ^ valueString v ^ tail vs
      end
```

T.6 Type checking

T.6.1 *Functions on the static environment*

Looking up values

S507c. ⟨*environments for Molecule's defined names* S507c⟩≡ (S501a)

```
(*
fun whatkind (COMPVAL _) = "a value"
  | whatkind (COMPTY _)  = "an ordinary type"
  | whatkind (COMPOVL _) = "an overloading group"
  | whatkind (COMPMOD _) = "a module"
*)

fun whatcomp (COMPVAL _) = "a value"
  | whatcomp (COMPABSTY _) = "an abstract type"
  | whatcomp (COMPMANTY _) = "a manifest type"
  | whatcomp (COMPMOD _) = "a module"

fun whatdec (ENVVAL _) = "a value"
  | whatdec (ENVMANTY _) = "a manifest type"
  | whatdec (ENVOVLN _) = "an overloaded name"
  | whatdec (ENVMOD _) = "a module"
```

```
  | whatdec (ENVMODTY _) = "a module type"

fun bigdec (ENVOVLN taus) = "overloaded at " ^ Int.toString (length taus) ^
                            " : [" ^ commaSep (map typeString taus) ^ "]"
  | bigdec d = whatdec d

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 m
  | compString (ENVMODTY _) = "a module type"



(*
fun findModty (t, Gamma) =
  case find (t, Gamma)
    of MODTY mt => mt
     | COMPONENT c =>
         raise TypeError ("Used " ^ t ^ " to name a module type, but " ^ t ^
                          " is " ^ whatkind c)
*)
```

S508a. ⟨*definitions of functions* `varTypeScheme`, `varType`, *and* `mutableVarType` S508a⟩≡ S508b ▷

```
fun varInfo (x, env) =
  case find (x, env)
    of STATIC_VAL info => info
     | _ => raise TypeError (x ^ " names a type, but a variable is expected")
```

S508b. ⟨*definitions of functions* `varTypeScheme`, `varType`, *and* `mutableVarType` S508a⟩+≡ ◁ S508a S508c ▷

```
fun varTypeScheme (x,E) = fst (varInfo (x, E))
```

S508c. ⟨*definitions of functions* `varTypeScheme`, `varType`, *and* `mutableVarType` S508a⟩+≡ ◁ S508b S508d ▷

```
fun varType (x, E) =
  case varTypeScheme (x, E)
    of FORALL ([], EXISTS _) =>
         raise TypeError (x ^ " names a type, but a variable is expected")
     | FORALL ([], tau) => tau
     | FORALL (_ :: _, _) =>
         raise TypeError (x ^ " must be instantiated before being used")
```

S508d. ⟨*definitions of functions* `varTypeScheme`, `varType`, *and* `mutableVarType` S508a⟩+≡ ◁ S508c

```
fun mutableVarType (x, E) =
  case varInfo (x, E)
    of (FORALL ([], tau), VARIABLE) => tau
     | (_, VARIABLE) => raise InternalError "polymorphic variable"
     | (_, _) => raise TypeError (x ^ " cannot be assigned to")
```

Looking up types

S508e. ⟨*internal functions* `asType` *and* `asTyvar`*, which check results of name lookup* S508e⟩≡ S509a ▷

```
fun asType (T, E) =
  case (find (T, E)
          handle NotFound _ => raise TypeError ("unknown type name " ^ T))
    of STATIC_VAL (FORALL ([], EXISTS _), _) => CONAPP (TYPART T, [])
     | STATIC_VAL (FORALL (_,  EXISTS _), _) =>
         raise TypeError
           (T ^ " is a type constructor and must be applied to type parameters")
     | STATIC_TYABBREV tau => tau
     | STATIC_TYVAR _ => TYVAR T
```

```
      | STATIC_VAL _ =>
            raise TypeError (T ^ " names a value, but a type is expected")
```

S509a. ⟨*internal functions* asType *and* asTyvar*, which check results of name lookup* S508e⟩+≡ ◁S508e

```
fun asTyvar (a, E) =
  case (find (a, E)
        handle NotFound _ =>
          raise TypeError ("type variable " ^ a ^ " is not in scope"))
    of STATIC_TYVAR _ => a
     | _ => raise InternalError (a ^ " in environment, but a type variable")
```


Stripping global variables

S509b. ⟨*Molecule's static environment* S509b⟩≡

```
stripvars : static env -> static env
```

```
fun stripvars E =
  let fun isVar (_, STATIC_VAL (_, VARIABLE)) = true
        | isVar _ = false
  in  List.filter (not o isVar) E
  end
```

T.6.2 Getting permission

A `return` is permissible if and only if P contains permission to return. In this case, `returnPermission` P returns `SOME` $[\tau_1, \ldots, \tau_n]$, where $[\tau_1, \ldots, \tau_n]$ gives the types of the values that may be returned. Function `yieldPermission` does the same for yielding.

S509c. ⟨*permissions* S509c⟩≡ S509d ▷

```
returnPermission : permissions -> ty list option
```

```
fun returnPermission [] = NONE
  | returnPermission (MAY_RETURN taus :: _) = SOME taus
  | returnPermission (_ :: permissions) = returnPermission permissions
```

S509d. ⟨*permissions* S509c⟩+≡ ◁S509c S509e ▷

```
yieldPermission : permissions -> ty list option
```

```
fun yieldPermission [] = NONE
  | yieldPermission (MAY_YIELD taus :: _) = SOME taus
  | yieldPermission (_ :: permissions) = yieldPermission permissions
```

Functions `mayBreak` and `mayContinue` tell whether breaking and continuing are permissible.

S509e. ⟨*permissions* S509c⟩+≡ ◁S509d

```
mayBreak    : permissions -> bool
mayContinue : permissions -> bool
```

```
val mayBreak =
  List.exists (fn MAY_BREAK => true | _ => false)
val mayContinue =
  List.exists (fn MAY_CONTINUE => true | _ => false)
```

T.6.3 Argument checking

In Molecule, there are three situations in which a list of expressions must have expected types:

- When arguments are passed to a function or iterator
- When results are provided by `return`
- When values are provided by `yield`

In any of these situations, if the types don't match, a diagnostic error message is produced by function argsTypeError.

S510a. ⟨*definition of* argsTypeError *for Molecule* S510a⟩≡

```
argsTypeError : string ->  want : ty list, got : ty list  -> 'a
```

```
fun argsTypeError what { want = ws , got = gs } =
  let fun raiseTheError (n, [], []) =
              raise InternalError "disappearing argsTypeError?!"
        | raiseTheError (n, want :: wants, got :: gots) =
            if eqType (want, got) then
              raiseTheError (n + 1, wants, gots)
            else
              raise TypeError ("argument " ^ intString n ^ " to " ^ what ^
                               " should have type " ^ typeString want ^
                               ", but it has type " ^ typeString got)
        | raiseTheError _ = raise InternalError "length mismatch"
  in  if length ws = length gs then
        raiseTheError (1, ws, gs)
      else
        raise TypeError (what ^ " expects " ^ countString ws "argument" ^
                         ", but it got " ^ intString (length gs))
  end
```


S510b. ⟨e_string *wanted* arrow *but got* arrow' S510b⟩≡

```
let val (wanted, got) = case arrow of FUNCTION => ("a function",  "iterator")
                                    | ITERATOR => ("an iterator", "function")
in  raise TypeError ("used " ^ got ^ " " ^ e_string ^ " as " ^ wanted)
end
```

S510c. ⟨*applied non-arrow* e_string S510c⟩≡

```
raise TypeError ("applied " ^ e_string ^ " of type " ^ typeString e's_tau ^
                 ", which is not a function type or iterator type")
```

T.6.4 Operator overloading

S510d. ⟨*Molecule's overloaded operators* S510d⟩≡ S511a ▷

```
val overloaded =   [ "+"
                   , "-"
                   , "*"
                   , "/"
                   , "mod"
                   , "power"
                   , "="
                   , "!="
                   , "<"
                   , ">"
                   , "<="
                   , ">="
                   , "similar?"
                   , "copy"
                   , "and"
                   , "or"
                   , "not"
                   , "negated"
                   , "print"
                   , "println"
                   , "at"
                   , "at-put"
                   ]
```

S511a. ⟨*Molecule's overloaded operators* S510d⟩+≡ ◁S510d S511b▷

```
fun isOverloaded name =
  List.exists (fn rator => name = rator) overloaded orelse
  String.isPrefix "get-" name orelse
  String.isPrefix "set-" name
```

S511b. ⟨*Molecule's overloaded operators* S510d⟩+≡ ◁S511a

```
fun maybeOverloadedName (VAR x) = if isOverloaded x then SOME x else NONE
  | maybeOverloadedName _       = NONE
```

T.6.5 Compatibility of a cluster with a previously defined interface

S511c. ⟨*if* x *is in* E *as a cluster interface, fail unless* sigma *is compatible* S511c⟩≡

```
case (SOME (varInfo (x, E)) handle _ => NONE)
  of SOME (sigma', CLUSTER_INTERFACE) =>
       checkInterfaceCompatibility { cluster = x, want = sigma', have = sigma }
   | _ => ()
```

S511d. ⟨*functions to check equality and compatibility of Molecule types* S511d⟩≡

```
fun checkInterfaceCompatibility
      { cluster = x, want = FORALL (aCws, tau), have = FORALL (aCws', tau') } =
  let fun fail ss = raise TypeError (String.concat ("in cluster " :: x :: ", " :: ss))
      ⟨internal function checkParam S512a⟩
      fun badLengths () =
        fail ["interface has ", countString aCws "type parameter", " but ",
              "implementation has ", countString aCws' "type parameter"]

      val _ = if length aCws <> length aCws' then badLengths () else ()
      val _ = ListPair.appEq checkParam (aCws, aCws')
                handle ListPair.UnequalLengths => badLengths ()

      fun checkTypes (EXISTS (XRECORDTY exports), EXISTS (XRECORDTY exports')) =
            let fun checkExport (x, tau) =
                      if eqType (find (x, exports), tau)
                         handle NotFound x =>
                           fail ["the implementation exports operation ", x,
                                 ", which is not exported by the interface"]
                      then
                        ()
                      else
                        fail ["the interface exports ", x, " with type ",
                              typeString (find (x, exports)), ", but the implementation ",
                              "exports ", x, " with type ", typeString tau]
                fun ensureNotMissing (x, tau) =
                      ignore (find (x, exports'))
                      handle NotFound x =>
                        fail ["the interface exports operation ", x,
                              ", which is not exported by the implementation"]
            in  ( app checkExport exports'
                ; app ensureNotMissing exports
                )
            end
        | checkTypes (EXISTS _ , ARROWTY _) =
            raise TypeError (x ^ " names a cluster interface and cannot be " ^
                                 "redefined as a routine")
        | checkTypes (tau, tau') =
            if eqType (tau, tau') then
              ()
```

```
            else
              fail ["interface exports type ", typeString tau, ", but ",
                    "implementation exports type ", typeString tau']
        val _ = checkTypes (tau, tau')
    in  ()
    end
```

S512a. ⟨*internal function* checkParam S512a⟩≡ (S511d)

```
  fun checkParam ((alpha, HAS Cw), (alpha', HAS Cw')) =
    let fun has (x, tau) =
          "[" ^ alpha ^ " has [" ^ x ^ " : " ^ typeString tau ^ "]]"
        fun checkConstraint (x, tau) =
          if (eqType (find (x, Cw), tau)
              handle NotFound x =>
                fail ["the implementation's where clause requires ", has (x, tau),
                      ", which the interface does not"])
          then
            ()
          else
            fail ["the interface's where clause requires ", has (x, find (x, Cw)),
                  ", but the implementation requires ", has (x, tau)]
        fun ensureNotMissing (x, tau) =
          ignore (find (x, Cw'))
          handle NotFound x =>
            fail ["the interface's where clause requires ", has (x, tau),
                  ", which the implementation does not"]
    in  if alpha <> alpha' then
          fail ["type parameter is called ", alpha, " in the interface but ",
                alpha', " in the implementation"]
        else
          ( app checkConstraint Cw'
          ; app ensureNotMissing Cw
          )
    end
```

S512b. ⟨*legacy test cases* S512b⟩≡ S512c ▷

```
  -> (cluster interface interface-routine-fail [exports [bar : ( -> interface-routine-fail)]]
  cluster interface-routine-fail
  -> (define interface-routine-fail ([n : int] -> bool) (return #t))
  type error: interface-routine-fail names a cluster interface and cannot be redefined as a r
```

S512c. ⟨*legacy test cases* S512b⟩+≡ ◁ S512b S512d ▷

```
  -> (cluster interface bad-interface [exports] (type rep null))
  type error: cluster interface bad-interface must not have any definitions
  -> (cluster interface mismatch1 [exports])
  -> (cluster ['a] mismatch1 [exports] (type rep null))
  type error: in cluster mismatch1, interface has 0 type parameters but implementation has 1
  -> (cluster interface ['b 'a] mismatch2 [exports])
  -> (cluster           ['a] mismatch2 [exports] (type rep null))
  type error: in cluster mismatch2, interface has 2 type parameters but implementation has 1
  -> (cluster interface ['b 'a] mismatch3 [exports])
  -> (cluster           ['a 'b] mismatch3 [exports] (type rep null))
  type error: in cluster mismatch3, type parameter is called 'b in the interface but 'a in th
```

S512d. ⟨*legacy test cases* S512b⟩+≡ ◁ S512c S513a ▷

```
  -> (cluster interface ['a] mm4 [exports])
  -> (cluster           ['a where ['a has [nifty? : ('a -> int)]]]
      mm4 [exports] (type rep null))
  type error: in cluster mm4, the implementation's where clause requires ['a has [nifty? : ('
```

```
-> (cluster interface ['a where ['a has [nifty? : ('a -> bool)]]] mm5 [exports])
-> (cluster           ['a where ['a has [nifty? : ('a -> int)]]]
    mm5 [exports] (type rep null))
type error: in cluster mm5, the interface's where clause requires ['a has [nifty? : ('a -> bool)]],
-> (cluster interface ['a where ['a has [nifty? : ('a -> bool)]]] mm6 [exports])
-> (cluster           ['a]
    mm6 [exports] (type rep null))
type error: in cluster mm6, the interface's where clause requires ['a has [nifty? : ('a -> bool)[redacted],
```


S513a. ⟨*legacy test cases* S512b⟩+≡ ◁S512d S528b▷

```
-> (cluster interface mx0 [exports [ignore : (->)]])
-> (cluster           mx0 [exports [ignore : (->)]]
        (type rep null)
        (define ignore (->) (return)))
-> (cluster interface mx1 [exports])
-> (cluster           mx1 [exports [ignore : (->)]]
        (type rep null)
        (define ignore (->) (return)))
type error: in cluster mx1, the implementation exports operation ignore, which is not exported by th
-> (cluster interface mx2 [exports [ignore : (->)]])
-> (cluster           mx2 [exports]
        (type rep null)
        (define ignore (->) (return)))
type error: in cluster mx2, the interface exports operation ignore, which is not exported by the imp
-> (cluster interface mx3 [exports [ignore : (-> bool)]])
-> (cluster           mx3 [exports [ignore : (->)]]
        (type rep null)
        (define ignore (->) (return)))
type error: in cluster mx3, the interface exports ignore with type ( -> bool), but the implement[redacted]o
```

T.6.6 *Types for export records of primitive types*

S513b. ⟨*types of export records for array, record, sum, arrow, and primitive types* S513b⟩≡
⟨*infix functions for writing arrow types* S513c⟩
⟨*functions that give the types of operations for equality, similarity, copying, and printing* S513d⟩
⟨*types of the value parts of the primitive clusters* S514c⟩
⟨*types of the value parts of array, record, sum, and arrow types* S515d⟩

T.6.7 *Easy notation for function types*

S513c. ⟨*infix functions for writing arrow types* S513c⟩≡ (S513b)

```
infix 3 --> -->*
fun args -->  results = ARROWTY (args, FUNCTION, results)
fun args -->* results = ARROWTY (args, ITERATOR, results)
```

T.6.8 *Types of operations for equality, similarity, copying, and printing*

Type constructors can provide equality operations only if the underlying types also provide equality operations.

S513d. ⟨*functions that give the types of operations for equality, similarity, copying, and printing* S513d⟩≡ (S513b) S514a▷

`typeHas : static env -> ty * (name * ty) -> bool`

```
fun typeHas env (tau, (opname, optype)) =
  eqType (optype, find (opname, xrecordExports (tau, env)))
  handle NotFound _ => false
```

S514a. ⟨*functions that give the types of operations for equality, similarity, copying, and printing* S513d⟩+≡ (S513b

```
eqSimCopyExports : static env -> mutability -> ty -> ty list -> (name * ty) list
```

```
fun basetype x = CONAPP (TYPART x, []) : ty
val booltype = basetype "bool"

fun eqSimCopyExports env mutability tau argtypes =
  let val bool = booltype
      fun cmptype tau = [tau, tau] --> [bool]
      fun cpytype tau = [tau] --> [tau]

      fun whenAllArgsHave (opname, typeFrom) any =
        if List.all (fn tau => typeHas env (tau, (opname, typeFrom tau))) argtypes then
          SOME any
        else
          NONE
      val always = SOME

      val cmp = cmptype tau
      val cpy = cpytype tau
  in  case mutability
        of IMMUTABLE =>
             List.mapPartial id
               [ whenAllArgsHave ("=",        cmptype) ("=",    cmp)
               , whenAllArgsHave ("=",        cmptype) ("!=",   cmp)
               , whenAllArgsHave ("similar?", cmptype) ("similar?",  cmp)
               , whenAllArgsHave ("copy",     cpytype) ("copy",     cpy)
               ]
         | MUTABLE =>
             List.mapPartial id
               [ always                                ("=",    cmp)
               , always                                ("!=",   cmp)
               , whenAllArgsHave ("similar?", cmptype) ("similar?",  cmp)
               , whenAllArgsHave ("=",        cmptype) ("similar1?", cmp)
               , always                                ("copy1",    cpy)
               , whenAllArgsHave ("copy",     cpytype) ("copy",     cpy)
               ]
  end
```

SPECIAL CASES WORTH NOTING.

S514b. ⟨*functions that give the types of operations for equality, similarity, copying, and printing* S513d⟩+≡ (S513b

```
fun baseEqSimCopyExports mutability tau = eqSimCopyExports emptyEnv mutability tau []
fun printExports tau = [ ("print",   [tau] --> [])
                       , ("println", [tau] --> [])
                       ]
fun immutableExports tau = baseEqSimCopyExports IMMUTABLE tau @ printExports tau
```

T.6.9 Types of the exported operations of primitive clusters

Exported operations refer to the type.

S514c. ⟨*types of the value parts of the primitive clusters* S514c⟩≡ (S513b) S514d ▷

Exported operations of type `bool`

S514d. ⟨*types of the value parts of the primitive clusters* S514c⟩+≡ (S513b) ◁ S514c S515a ▷

```
val boolXrecordType =
  [ ("and", [booltype, booltype] --> [booltype])
```

```
  , ("or",  [booltype, booltype] --> [booltype])
  , ("not", [booltype]        --> [booltype])
  ] @
  baseEqSimCopyExports IMMUTABLE booltype @
  printExports booltype
```

S515a. ⟨*types of the value parts of the primitive clusters* S514c⟩+≡ (S513b) ◁S514d S515b▷

```
val nulltype = basetype "null"
val nullXrecordType  = immutableExports nulltype
```


S515b. ⟨*types of the value parts of the primitive clusters* S514c⟩+≡ (S513b) ◁S515a S515c▷

```
val inttype = basetype "int"
val intXrecordType =
  [ ("+",    [inttype, inttype] --> [inttype])
  , ("-",    [inttype, inttype] --> [inttype])
  , ("*",    [inttype, inttype] --> [inttype])
  , ("/",    [inttype, inttype] --> [inttype])
  , ("negated", [inttype] --> [inttype])
  , ("mod", [inttype, inttype] --> [inttype])
  , ("power", [inttype, inttype] --> [inttype])
  , ("max", [inttype, inttype] --> [inttype])
  , ("min", [inttype, inttype] --> [inttype])
  , ("abs", [inttype] --> [inttype])
  , ("from-to-by", [inttype, inttype, inttype] -->* [inttype])
  , ("from-to",    [inttype, inttype] -->* [inttype])
  , ("<",     [inttype, inttype] --> [booltype])
  , (">",     [inttype, inttype] --> [booltype])
  , ("<=",    [inttype, inttype] --> [booltype])
  , (">=",    [inttype, inttype] --> [booltype])
  , ("printu", [inttype] --> [])
  ] @
  immutableExports inttype
```

S515c. ⟨*types of the value parts of the primitive clusters* S514c⟩+≡ (S513b) ◁S515b

```
val symtype = basetype "sym"
val symXrecordType = [("hash", [symtype] --> [inttype])] @ immutableExports symtype
```

T.6.10 *Types of value parts of array types*

I omit CLU's `trim` primitive because it's too hard to explain.

S515d. ⟨*types of the value parts of array, record, sum, and arrow types* S515d⟩≡ (S513b) S516▷

```
arrayXrecordType  : (mutability * ty) * static env -> ty env
```

```
fun arrayXrecordType ((mutability, elem), env) =
      let val array = ARRAYTY (mutability, elem)
          val both = SOME
          val (m, i) = case mutability
                         of MUTABLE => (SOME, fn _ => NONE)
                          | IMMUTABLE => (fn _ => NONE, SOME)
      in  List.mapPartial id
          [ both ("new",      []      --> [array])
          , m    ("create", [inttype] --> [array])

          , both ("bottom",   [array] --> [elem])
          , both ("top",      [array] --> [elem])
          , m    ("low",    [array] --> [inttype])
          , m    ("high",   [array] --> [inttype])
          , both ("size",     [array] --> [inttype])
```

```
            , both ("empty?",   [array] --> [booltype])

            , both ("at",       [array, inttype] --> [elem])
            , m    ("at-put", [array, inttype, elem] --> [])
            , i    ("replace", [array, inttype, elem] --> [array])

            , m    ("addl",   [array, elem] --> [])
            , m    ("addh",   [array, elem] --> [])
            , m    ("reml",   [array] --> [elem])
            , m    ("remh",   [array] --> [elem])
            , i    ("addl",   [array, elem] --> [array])
            , i    ("addh",   [array, elem] --> [array])
            , i    ("reml",   [array] --> [array])
            , i    ("remh",   [array] --> [array])

            , m    ("set-low", [array, inttype] -->          [])

            , m    ("fill",      [inttype, inttype, elem] --> [array])
            , m    ("fill-copy", [inttype, inttype, elem] --> [array])
            , i    ("fill",   [inttype, elem] --> [array])

            , both ("elements", [array] -->* [elem])
            , both ("indices",  [array] -->* [inttype])

            , i    ("subseq", [array, inttype, inttype] --> [array])
            , i    ("e2a",    [elem] --> [array])
            , i    ("append", [array, array] --> [array])
            , i    ("ia2ma", [array] --> [ARRAYTY (MUTABLE, elem)])
            , i    ("ma2ia", [ARRAYTY (MUTABLE, elem)] --> [array])
            ]
            @ eqSimCopyExports env mutability array [elem]
            @ printExports array
        end
```

T.6.11 Types of value parts of record types

S516. ⟨*types of the value parts of array, record, sum, and arrow types* S515d⟩+≡ (S513b) ◁S515d S517a▷

```
        recordXrecordType  : (mutability * (name * ty) list) * static env -> ty env
  fun recordXrecordType ((mutability, fields), env) =
        let val record = RECORDTY (mutability, fields)
            fun fops f = map f fields
            val all = fops (fn (x, tau) => ("get-" ^ x, [record] --> [tau]))
            val special =
              case mutability
                of MUTABLE =>
                     fops (fn (x, tau) => ("set-" ^ x, [record,tau] --> [])) @
                     [ ("mr_gets_mr", [record, record] --> [])
                     , ("mr_gets_ir", [record, RECORDTY (IMMUTABLE, fields)] --> [])
                     ]
                 | IMMUTABLE =>
                     fops (fn (x, tau) => ("replace-" ^ x, [record,tau] --> [record])) @
                     [ ("ir2mr", [record] --> [RECORDTY (MUTABLE, fields)])
                     , ("mr2ir", [RECORDTY (MUTABLE, fields)] --> [record])
                     ]
        in  all @ special @ eqSimCopyExports env mutability record (map snd fields)
                @ printExports record
```

```
        end
```

T.6.12 Types of value parts of sum types

S517a. ⟨*types of the value parts of array, record, sum, and arrow types* S515d⟩+≡ (S513b) ◁S516 S517b▷

```
oneofXrecordType  : (mutability * (name * ty) list) * static env -> ty env
fun oneofXrecordType ((mutability, variants), env) =
      let val oneof = ONEOFTY (mutability, variants)
          fun vops f = map f variants
          val all = vops (fn (x, tau) => ("make-" ^ x, [tau] --> [oneof])) @
                    vops (fn (x, tau) => ("is-" ^ x ^ "?", [oneof] --> [booltype])) @
                    vops (fn (x, tau) => ("value-" ^ x, [oneof] --> [tau]))
          val special =
            case mutability
              of MUTABLE =>
                   vops (fn (x, tau) => ("change-" ^ x, [oneof,tau] --> [])) @
                   [ ("mo_gets_mo", [oneof, oneof] --> [])
                   , ("mo_gets_io", [oneof, ONEOFTY (IMMUTABLE, variants)] --> [])
                   ]
               | IMMUTABLE =>
                   [ ("io2mo", [oneof] --> [ONEOFTY (MUTABLE, variants)])
                   , ("mo2io", [ONEOFTY (MUTABLE, variants)] --> [oneof])
                   ]
      in  all @ special @ eqSimCopyExports env mutability oneof (map snd variants)
              @ printExports oneof
      end
```


T.6.13 Types of value parts of arrow types

S517b. ⟨*types of the value parts of array, record, sum, and arrow types* S515d⟩+≡ (S513b) ◁S517a

```
fun arrowXrecordType (spec, _) =
  let val tau = ARROWTY spec
  in  baseEqSimCopyExports IMMUTABLE tau @ printExports tau
  end
```

T.7 Lexical analysis and parsing

S517c. ⟨*lexical analysis and parsing for Molecule, providing* `filexdefs` *and* `stringsxdefs` S517c⟩≡ (S501a)

```
⟨lexical analysis for Molecule S517d⟩
fun 'a parseAt at p = at <$> @@ p
⟨parsers for Molecule tokens S519a⟩
val booltok = pzero  (* depressing *)
⟨parsers for μML value constructors and value variables generated automatically⟩
⟨parsers and parser builders for formal parameters and bindings S375a⟩
val tyvar = sat (fn _ => false) name  (* must have a monomorphic type *)
⟨parser builders for typed languages S387a⟩
⟨parsers and xdef streams for Molecule S519c⟩
⟨shared definitions of filexdefs and stringsxdefs S254c⟩
```

`<$>`	S263b
`name`	S519a
`pzero`	S264b
`sat`	S266a

S517d. ⟨*lexical analysis for Molecule* S517d⟩≡ (S517c) S518a▷

```
datatype pretoken = QUOTE
                  | INT      of int
                  | RESERVED of string
                  | DOTTED   of string * string list
                         (* name, possibly followed by dotted selection *)
```

```
                    | DOTNAMES of string list (* .x.y and so on *)
type token = pretoken plus_brackets
```

S518a. ⟨*lexical analysis for Molecule* S517d⟩+≡ (S517c) ◁S517d S518b▷

```
fun pretokenString (QUOTE)       = "'"
  | pretokenString (INT  n)      = intString n
  | pretokenString (DOTTED (s, ss))  = separate ("", ".") (s::ss)
  | pretokenString (DOTNAMES ss)= (concat o map (fn s => "." ^ s)) ss
  | pretokenString (RESERVED x) = x
val tokenString = plusBracketsString pretokenString
```


Every character is either a symbol, an alphanumeric, a space, or a delimiter.

S518b. ⟨*lexical analysis for Molecule* S517d⟩+≡ (S517c) ◁S518a

`mclToken : token lexer`

```
local
  val isDelim = fn c => isDelim c orelse c = #"."
  ⟨functions used in all lexers S374c⟩
  val reserved =
    [ ⟨words reserved for Molecule types S519b⟩
    , ⟨words reserved for Molecule expressions S521a⟩
    , ⟨words reserved for Molecule definitions S523⟩
    ]
  fun isReserved x = member x reserved
  datatype part = DOT | NONDELIMS of string
  val nondelims = (NONDELIMS o implode) <$> many1 (sat (not o isDelim) one)
  val dot       = DOT <$ eqx #"." one
  fun dottedNames things =
    let exception Can'tHappen
        fun preDot (ss', DOT :: things)     = postDot (ss', things)
          | preDot (ss', nil)               = OK (rev ss')
          | preDot (ss', NONDELIMS _ :: _) = raise Can'tHappen
        and postDot (ss', DOT :: _) = ERROR "A qualified name may not contain consecutive d
          | 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

  fun reserve (token as DOTTED (s, [])) =
        if isReserved s then
          RESERVED s
        else
          token
    | reserve token = token

in
  val mclToken =
    whitespace *>
    bracketLexer (  QUOTE   <$  eqx #"'" one
                <|> INT     <$> intToken isDelim
                <|> reserve <$> (dottedNames <$>! many1 (nondelims <|> dot))
                <|> noneIfLineEnds
                 )
```

```
end
```

S519a. ⟨*parsers for Molecule tokens* S519a⟩≡ (S517c)

```
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) <$>? pretoken
val name  = (fn (DOTTED (x, []))   => SOME x  | _ => NONE) <$>? 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 any_name = name

val arrow = eqx "->" reserved <|> eqx "--m->" reserved

val showErrorInput = (fn p => showErrorInput tokenString p)
```

T.8 Parsing

S519b. ⟨*words reserved for Molecule types* S519b⟩≡ (S518b S521b)

```
"->", ":"
```

S519c. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩≡ (S517c) S519d ▷

```
fun kw keyword = eqx keyword reserved
fun usageParsers ps = anyParser (map (usageParser kw) ps)
```

S519d. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁ S519c S519e ▷

```
fun getkeyword (usage:string) = (one *> one *> one) (lexLineWith mclToken usage)
```

S519e. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁ S519d S519f ▷

```
fun wrap   what = wrapAround tokenString what
fun wrap_ what p = p
```

S519f. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁ S519e S519g ▷

```
fun showParsed show p =
  let fun diagnose a = (eprintln ("parsed " ^ show a); a)
  in  diagnose <$> p
  end

fun showParsed_ show p = p
```

S519g. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁ S519f S519h ▷

```
fun bracketOrFail (_, p) =
  let fun matches (_, l) a (loc, r) =
        if l = r then OK a
        else errorAt (leftString l ^ " closed by " ^ rightString r) loc
  in  matches <$> left <*> p <*>! right
  end
```

S519h. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁ S519g S520a ▷

```

fun addDots p xs = foldl (fn (x, p) => PDOT (p, x)) p xs
fun dotsPath (loc, (x, xs)) = addDots (PNAME (loc, x)) xs
fun path tokens =
  ( dotsPath <$> @@ dotted
  <|>
      addDots <$>
        bracketKeyword
           (kw "@m", "(@m name path ...)", curry PAPPLY <$> (PNAME <$> @@ name)
              <*> (dotnames <|> pure [])
```

```
    ) tokens

  fun mkTyex br tokens =
    let val ty = wrap_ "inner type" (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"
        val parser =
             TYNAME <$> path
          <|> br
                ( "(ty ty ... -> ty)"
                ,  arrows <$> many ty <*>! many (kw "->" *> many ty)
                )
    in  parser (* curry TYEX_AT () <$> @@ parser *)
    end tokens
  val tyex = wrap_ "tyex" (mkTyex (showErrorInput o bracket)) : tyex parser
  val liberalTyex = mkTyex bracketOrFail
```

XXX NEED TO HANDLE CONVAL

S520a. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S519h S520b▷

```
  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 =  sat (isVcon o vconLast) (dottedVcon <$> dotted)
          <|> PNAME <$> bare_vcon
          <|> (fn (loc, (x, xs)) => errorAt ("Expected value constructor, but got name " ^
                                  foldl (fn (x, p) => p ^ "." ^ x) x xs) loc)
              <$>! @@ dotted

  fun pattern tokens =  (
                  WILDCARD    <$  eqx "_" vvar
        <|>       PVAR        <$> vvar
        <|> curry CONPAT      <$> vcon <*> pure []
        <|> bracket ( "(C x1 x2 ...) in pattern"
                    , curry CONPAT <$> vcon <*> many pattern
                    )
         ) tokens
```

NO COMPONENTS AT TOP LEVEL!

S520b. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S520a S521b▷

```
                                                exptable : exp parser -> exp parser
                                                exp      : exp parser
  fun badReserved r =
    ERROR ("reserved word '" ^ r ^ "' where name was expected")

  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

val atomicExp =  VAR           <$> path
             <|> badReserved <$>! reserved
             <|> dotnames <!> "a qualified name may not begin with a dot"
             <|> LITERAL <$> NUM <$> int
             <|> VCONX <$> vcon
             <|> quote *> (LITERAL <$> quotelit)

fun bindTo exp = bracket ("[x e]", pair <$> name <*> exp)
```

S521a. ⟨*words reserved for Molecule expressions* S521a⟩≡ (S518b)

```
"@m", "if", "&&", "||", "set", "let", "let*", "letrec", "case", "lambda",
"val", "set", "while", "begin", "error",
"when", "unless", "assert"
(* , "assert" *)
```

S521b. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S520b S522a▷

```
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 na



fun smartBegin [e] = e
  | smartBegin es = BEGIN es

fun exptable exp =
  let val zero = LITERAL (NUM 0)
      fun single binding = [binding]
      fun badReserved words =
        let fun die w = ERROR ("while trying to parse an expression, I see " ^
                               "reserved word " ^ w ^
                               "... did you misspell a statement keyword earlie
        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 choice   = bracket ("[pattern exp]", pair <$> pattern <*> exp)
      val body = smartBegin <$> many1 exp
      val nothing = pure (BEGIN [])

      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 ||"

     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 <*> not
     , ("(unless e1 e ...)",      curry3 IFX        <$> exp <*> nothing <*>
```

```
    , ("(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 <$> tbindings <*> body)
    , ("(case exp (pattern exp) ...)", curry CASE <$> exp <*> many choice)
    , ("(lambda ([x : ty] ...) body)", lambda <$> @@ (lformals : (name * tyex) list parser
    , ("(&& e ...)",               cand <$> many1 exp)
    , ("(|| e ...)",               cor  <$> many1 exp)
    , ("(assert e)",
       curry3 IFX <$> exp <*> nothing <*> pure (ERRORX [LITERAL (SYM "assertion-failure")]
    , ("(quote sx)",               LITERAL <$> quotelit)
    ]
   <|> badReserved [⟨words reserved for Molecule types S519b⟩,
                    ⟨words reserved for Molecule definitions S523⟩]
  end
```

S522a. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S521b S522b▷

```
fun applyNode f args = APPLY (f, args, ref notOverloadedIndex)
fun exp tokens = showParsed_ expString (parseAt EXP_AT replExp) tokens
and replExp tokens = showErrorInput
       (  (* component here only if type with reserved word *)
          atomicExp
      <|> exptable exp
      <|> leftCurly <!> "curly brackets are not supported"
      <|> left *> right <!> "empty application"
      <|> bracket("function application", applyNode <$> exp <*> many exp)
  ) tokens


val replExp = showParsed_ expString (parseAt EXP_AT replExp)
```

S522b. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S522a S524a▷

```
fun formalWith whatTy aTy =
  bracket ("[x : " ^ whatTy ^ "]", pair <$> name <* kw ":" <*> aTy)

val formal = formalWith "ty" tyex
```

```
decl : (name * decl) parser
locmodformal : (name located * modtyex located) parser
modformal    : (name * modtyex) parser
modtype      : modtyex parser
```

```
fun prightmap f (x, a) = (x, f a)
fun crightmap f x a = (x, f a)

fun recordOpsType tyname (loc, formals : (name * tyex) list) =
  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

fun recordModule (loc, name) tyname (formals : (name * tyex) list) =
  let val t = TYNAME (PNAME (loc, tyname))
      val vcon = "make-" ^ name ^ "." ^ tyname
      val conpat = CONPAT (PNAME vcon, map (PVAR o fst) formals)
```

```
        val conname = name ^ ".make"
        fun setname x = "set-" ^ x ^ "!"
        fun var x = VAR (PNAME (loc, x))
        val conval =
          LAMBDA (formals, APPLY (VCONX (PNAME vcon), map (var o fst) formals, ref notOverloadedIndex)
        fun getter n =
          (LAMBDA ([("r", t)],
                   CASE (var "r", [(conpat, var (fst (List.nth (formals, n))))])))
        fun setter n =
          (LAMBDA ([("the record", t), ("the value", snd (List.nth (formals, n)))],
                   CASE (var "the record",
                         [(conpat, SET (fst (List.nth (formals, n)), var "the valu
        val modty = recordOpsType tyname (loc, formals)

        fun prim (x, f) = VAL (x, f)
        val indices = List.tabulate (length formals, id)
        val components =
          DATA (tyname, [(vcon, FUNTY (map snd formals, t))]) ::
          prim ("make", conval) ::
          ListPair.mapEq (fn ((x,_), i) => prim (x, getter i)) (formals, indices)
          ListPair.mapEq (fn ((x,_), i) => prim (setname x, setter i)) (formals, i
    in  MODULE (name, MSEALED (modty, map (fn d => (loc, d)) components))
    end


  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 <$> name <*> @@ lfor
    ]
    <|> MTNAMEDX <$> name
    <|> bracket ("([A : modty] ... --m-> modty)",
                 curry MTARROWX <$> many locmodformal <*> kw "--m->" *> @@ modtype
    ) tokens
```

S523. ⟨*words reserved for Molecule definitions* S523⟩≡ (S518b S521b)

```
  ":",
  "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"
```

S524a. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S522b S524b▷

```
val tyex : tyex parser = tyex
```

Value variables and value constructors.

S524b. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S524a S524c▷

```
fun wantedVcon (loc, x) = errorAt ("expected value constructor, but got name " ^ x) loc
fun wantedVvar (loc, x) = errorAt ("expected variable name, but got value constructor " ^ x

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
                   )
```

Goal for definitions:

1. Extended definitions
2. Definition keywords (which cover the binding statements)
3. Statement keywords
4. Expressions of which function application turns into a call statement

S524c. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S524b S525▷

`def : def parser`

```
val defFwd    = ref (forward "def" : def parser)
fun def arg    = !defFwd arg

fun def tokens =
  let val returnTypes = bracket("[ty ...]", many tyex) <|> pure []
  in  showErrorInput (!defFwd)
  end tokens

val def = wrap_ "def" def : def parser

val defbasic : baredef parser =
  let (* parser for binding to names *)
      val formals = lformals : (name * tyex) list parser
  (*    val formals = vvarFormalsIn "define" *)

      (* parsers for clausal definitions, a.k.a. define* *)
(*
      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)
*)
      (* definition builders used in all parsers *)
      fun flipPair tx c = (c, tx)
```

```
      (* definition builders that expect to bind names *)
      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 = smartBegin <$> many1 exp

  in  usageParsers
      [ ("(define type f (args) body)",
                                      define <$> tyex <*> name <*> @@ lformals <
      , ("(val x e)",                 curry VAL <$> vvar <*> exp)
      , ("(val-rec [x : type] e)",    valrec <$> formal <*> exp)

      , ("(data t [vcon : ty] ...)",
         wrap_ "data definition" (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 : (name * moddef) pa
                      <|> (sealedWith MPATHSEALED <$> modformal <*> path : (name      r)
                      <|> (sealedWith MSEALED <$> modformal <*> many def : (name     r)
                       ))

      , ("(generic-module [M : T] defs)",
            let fun strip ((_, m), (_, t)) = (m, t)
                fun gen ((loc, M), (loc', T)) defs =
                  case T
                    of MTARROWX (formals, result) =>
                         OK (GMODULE (M, map strip formals, MSEALED (snd result,
                     | _ => 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 <*> formals)
      , ("(overload qname ...)", OVERLOAD <$> many path)
      ]
     <|> QNAME <$> path
     <|> EXP <$> exp : baredef parser
  end

val _ = defFwd := @@ defbasic
```

S525. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S524c S526a▷

```
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)
  ]
```

S526a. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S525 S526b▷

```
fun filenameOfDotted (x, xs) = separate ("", ".") (x :: xs)
val xdeftable = usageParsers
  [ ("(use filename)", (USE o filenameOfDotted) <$> dotted)
  ]
```

S526b. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S526a S526c▷

```
val xdef =  TEST <$> testtable
        <|>          xdeftable
        <|> DEF <$>  def
        <|> badRight "unexpected right bracket"
        <?> "definition"
```

S526c. ⟨*parsers and* `xdef` *streams for Molecule* S519c⟩+≡ (S517c) ◁S526b

```
val xdefstream =
  interactiveParsedStream (mclToken, xdef)
```

T.9 Unit testing

S526d. ⟨*definition of* `testIsGood` *for Molecule* S526d⟩≡ (S501b) S526e▷

```
fun comparisonIndex env tau =
  let val wanted = FUNTY ([tau, tau], booltype)
      val index  =
        case find ("=", env)
          of ENVOVLN taus =>
               (case resolveOverloaded ("=", tau, taus)
                  of OK (compty, i) =>
                       if eqType (compty, wanted) then OK i
                       else (ERROR o String.concat)
                            ["on type ", typeString tau, " operation = has type ",
                             typeString compty]
                   | ERROR msg => ERROR msg)
           | _ => ERROR "operator = is not overloaded, so I can't check-expect"
  in  index
  end
```

S526e. ⟨*definition of* `testIsGood` *for Molecule* S526d⟩+≡ (S501b) ◁S526d

```
fun noTypeError f x k =
  (f x; true) handle TypeError msg => failtest (k msg)

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 S384d⟩
      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))     =
        noTypeError elabty (t, E)
        (fn msg => ["In (check-type ", expString e, " " ^ tyexString t, "),
    | 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]

  fun deftystring d =
    let val comps = List.mapPartial asComponent (elabd (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


  fun outcome e = withHandlers (fn () => OK (eval (e, rho))) () (ERROR o str
```
⟨*definition of* `asSyntacticValue` *for Molecule* S528a⟩
⟨*shared* `whatWasExpected` S245b⟩
⟨*shared* `checkExpectPassesWith`, *which calls* `outcome` S245c⟩
⟨*shared* `checkAssertPasses` *and* `checkErrorPasses`, *which call* `outcome` S246a⟩

```
  fun checkExpectPasses (c, e) =
    let val i = case comparisonIndex E (ty c)
                  of OK i => i
                   | ERROR _ => raise InternalError "overloaded = in check-e
        val eqfun =
          case !(find ("=", rho))
            of ARRAY vs => (Array.sub (vs, i)
                            handle _ => raise InternalError "overloaded subs
             | _ => raise InternalError "overloaded = not array"

        fun testEqual (v1, v2) =
          case eval (APPLY (LITERAL eqfun, [LITERAL v1, LITERAL v2], ref not
            of CONVAL (PNAME "#t", []) => true
             | _ => false

    in  checkExpectPassesWith testEqual (c, e)
    end

  fun checkMtypePasses (pathx, mtx) =
    let val path = txpath (pathx, E)
        val principal = strengthen (findModule (pathx, E), path)
        val mt = elabmt ((mtx, path), E)
val () = if true then () else
         ( app print ["principal MT   = ", mtString principal, "\n"]
         ; app print ["supertype      = ", mtString mt, "\n"]
         ; app print ["supertype path = ", pathString path, "\n"]
         )
```

```
            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]
```

⟨*shared* checkTypePasses *and* checkTypeErrorPasses*, which call* ty S384b⟩

```
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
```

S528a. ⟨*definition of* asSyntacticValue *for Molecule* S528a⟩≡ (S526e)

`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
```

T.10 Miscellaneous error messages

S528b. ⟨*legacy test cases* S512b⟩+≡ ◁S513a S530b▷

```
-> (define multiple-tags ([x : bad-tags-type] -> )
      (tag-case x
         (a (return))
         (b (return))
         (b (return))))
type error: tag b used multiple times in tag-case
-> (define redundant-others ([x : bad-tags-type] -> )
      (tag-case x
         (a (return))
         (b (return))
         (others (return))))
type error: 'others' case in tag-case can never match
```

S528c. ⟨*utility functions* fieldsmap *and* fieldsort*, which operate on labeled values* S528c⟩≡

```
fun fieldsmap f = map (fn (x, a) => (x, f a))
```

S528d. ⟨*complain that* unmatched *tags are unmatched* S528d⟩≡

```
raise TypeError ("tag-case " ^ expString e ^ " does not match " ^
                 "these tags: " ^ spaceSep unmatched)
```

S528e. ⟨*complain that* e *doesn't have a sum type* S528e⟩≡

```
raise TypeError ("type of " ^ expString e ^ " passed to " ^ "tag-case is " ^
                 typeString (ty e) ^ ", which is not a one-of")
```

S528f. ⟨*fail unless* x'_i *is in both* all_variants *and* unmatched S528f⟩≡ S529a▷

S529a. ⟨*fail unless* `x'_i` *is in both* `all_variants` *and* `unmatched` S528f⟩+≡ ◁S528f

```
if not (isbound (x'_i, all_variants)) then
  raise TypeError ("type " ^ typeString (ty e) ^ " has no tag named " ^ x'_i)
else if not (member x'_i unmatched) then
  raise TypeError ("tag " ^ x'_i ^ " used " ^ "multiple times in tag-case")
else
  ()
```

S529b. ⟨*number of* `results` *doesn't match* `xs` S529b⟩≡

```
raise TypeError ("assignment has " ^ countString xs "variable" ^
                 " but call on the right produces " ^ countString results "result")
```

S529c. ⟨`y_i` *should have type* `tau_i` S529c⟩≡

```
raise TypeError ("tag " ^ x'_i ^ " declares " ^ y_i ^ " with type " ^ typeString tau'_i ^
                 ", but that tag carries type " ^ typeString tau_i)
```

S529d. ⟨*iterator's* `args` *don't match* `formals` S529d⟩≡

```
raise TypeError ("Iterator is expecting " ^ plural "parameter" formals ^
                 " of " ^ plural "type" formals ^ " " ^ typesString formals ^
                 ", but got actual " ^ plural "parameter" args ^ " of " ^
                 plural "type" args ^ typesString args)
```

S529e. ⟨*iterator's* `xs` *don't match* `results` S529e⟩≡

```
raise TypeError ("Iterator returns " ^ plural "result" results ^
                 " of " ^ plural "type" results ^ " " ^ typesString results ^
                 ", but assigns to " ^ plural "variable" xs ^
                 " of " ^ plural "type" xs ^ " " ^ typesString (map vartype xs))
```

S529f. ⟨`SETRESULTS` *bug* S529f⟩≡

```
raise BugInTypeChecking
  (expString (APPLY the_call) ^ " assigned to " ^ countString xs "argument" ^
   " but got " ^ countString vs "result")
```

S529g. ⟨*raise* `TypeError`*, showing* `unsatisfied` *constraints* S529g⟩≡

```
let fun single [_] = true
      | single _   = false
    fun unsatString (HASN'T (tau, opname, optype)) =
      typeString tau ^ " has " ^ "[" ^ opname ^ " : " ^ typeString optype ^ "]"
in  raise TypeError ("in " ^ typeString (CONAPP (TYPART T, taus)) ^
                     ", unsatisfied " ^ plural "constraint" unsatisfied ^
                     (if single unsatisfied then " " else ": ") ^
                        commaSep (map unsatString unsatisfied))
end
```

S529h. ⟨*type error:* `taus` *different length from* `alphas` S529h⟩≡

```
raise TypeError (T ^ " expects " ^ countString alphas "type parameter" ^
                 ", but got " ^ intString (length taus))
```

S529i. ⟨*type error:* `taus` *different length from* `formals` S529i⟩≡

```
raise TypeError (what ^ " expects " ^ countString formals "type parameter" ^
                 ", but got " ^ intString (length taus))
```

APPLY	S462a
CONVAL	S499d
LITERAL	S462a
optionList	S242a
VCONX	S462a

S529j. ⟨*desugaring was somehow inconsistent; fail* S529j⟩≡

```
raise InternalError ("in definition of " ^ x ^ ", expected type " ^
                     typeString tau ^ ", but got " ^ typeString tau' ^
                     " (should detect elsewhere)")
```

S529k. ⟨*complain that* `x` *is redefined* S529k⟩≡

```
let val asBound = find (x, E)
    val new = mkStatic a
    val asWhat = case asBound
                   of STATIC_VAL (FORALL (_, EXISTS _), CONSTANT) => "as a cluster"
```

```
                        | STATIC_VAL (_, CLUSTER_INTERFACE) => "as a cluster interface"
                        | STATIC_VAL (_, CONSTANT)          => "as a routine"
                        | STATIC_VAL (_, VARIABLE)          => "as a variable"
                        | STATIC_TYABBREV _                 => "as a type abbreviation"
                        | STATIC_TYVAR    _                 => "as a type variable"
  in raise TypeError
       ("redefinition of " ^ what ^ " " ^ x ^ ", which is already in scope " ^ asWhat)
  end
```

S530a. ⟨*if* `wheres` *constrains a non-*α_i *or constrains any operation multiple times, fail* S530a⟩≡

```
  let fun dieOnMultiplesOrStrays [] = ()
        | dieOnMultiplesOrStrays (WHERE (a, l, t) :: ws) =
            if List.exists (fn WHERE (a', l', _) => a = a' andalso l = l') ws then
              raise TypeError ("operation " ^ l ^ " on type parameter " ^
                               a ^ " is multiply constrained")
            else if not (member a alphas) then
              raise LeftAsExercise "where clause constrains outer type variable"
            else
              dieOnMultiplesOrStrays ws
  in  dieOnMultiplesOrStrays wheres
  end
```

S530b. ⟨*legacy test cases* S512b⟩+≡ ◁S528b S530c▷

```
  -> 3
  3 : int
  -> 'hello
  hello : sym
  -> (= 'hello 'daring)
  #f : bool
  -> (= #t #t)
  #t : bool
  -> 1
  1 : int
```

S530c. ⟨*legacy test cases* S512b⟩+≡ ◁S530b S531a▷

```
  -> (type ai1 (mutable array int))
  -> (val a1 (make-array-at 1 (mutable array int) 1 2 3 4 5))
  (mutable array [at 1] 1 2 3 4 5) : (mutable array int)
  -> (ai1$top a1)
  5 : int
  -> (ai1$reml a1)
  1 : int
  -> a1
  (mutable array [at 2] 2 3 4 5) : (mutable array int)
  -> (ai1$addl a1 99)
  -> a1
  (mutable array [at 1] 99 2 3 4 5) : (mutable array int)
  -> a1
  (mutable array [at 1] 99 2 3 4 5) : (mutable array int)
  -> (ai1$addh a1 33)
  -> a1
  (mutable array [at 1] 99 2 3 4 5 33) : (mutable array int)
  -> (ai1$addl a1 33)
  -> a1
  (mutable array [at 0] 33 99 2 3 4 5 33) : (mutable array int)
  -> (at a1 3)
  3 : int
```

S531a. ⟨*legacy test cases* S512b⟩+≡ ◁S530c S533b▷

```
-> (cluster ['a where ['a has [new : [ -> 'a]]]]
    wrap
            [exports [new : ( -> (wrap 'a))]]
      (type rep 'a)
      (define new ( -> (wrap 'a))
         (return (seal ('a$new)))))
cluster (wrap 'a)
-> (cluster void [exports] (type rep null))
cluster void
-> (type burble (mutable array void))
burble = (mutable array void)
-> (type clean (wrap (immutable array bool)))
clean = (wrap (immutable array bool))
-> burble$copy
type error: burble has no component named copy
-> (mutable array bool)$copy
<routine> : ((mutable array bool) -> (mutable array bool))
-> (type mab (mutable array bool))
mab = (mutable array bool)
-> mab$copy
<routine> : ((mutable array bool) -> (mutable array bool))
-> (type zorched (wrap void))
type error: in (wrap void), unsatisfied constraint void has [new : ( -> void)]
```


T.11 Printing stuff

S531b. ⟨*definition of* typeString *for Molecule types* S531b⟩≡ (S500b) S531c▷

```
fun modidentString (MODCON { printName = m, serial = 0 }) = m
  | modidentString (MODCON { printName = m, serial = k }) = m ^ "@{" ^ intString k ^ "}"
  | modidentString (MODTYPLACEHOLDER s) = "<signature: " ^ s ^ ">"

fun pathString' base =
  let fun s (PNAME a) = base a
        | s (PDOT (p, x)) = s p ^ "." ^ x
        | s (PAPPLY (f, args)) =
            String.concat ("(@m " :: s f ::
                           foldr (fn (a, tail) => " " :: s a :: tail) [")"] args)
  in  s
  end

fun pathString (PNAME a) = modidentString a
  | pathString (PDOT (PNAME (MODTYPLACEHOLDER _), x)) = x
  | pathString (PDOT (p, x)) = pathString p ^ "." ^ x
  | pathString (PAPPLY (f, args)) =
      String.concat ("(@m " :: pathString f ::
                     foldr (fn (a, tail) => " " :: pathString a :: tail) [")"] args)

(*val pathString = pathString' modidentString*)
val pathexString : pathex -> string = pathString' snd
```


S531c. ⟨*definition of* typeString *for Molecule types* S531b⟩+≡ (S500b) ◁S531b S532a▷

```
fun typeString' ps (TYNAME p) = ps p
  | typeString' ps (FUNTY (args, res)) =
      "(" ^ spaceSep (map (typeString' ps) args) ^ " -> " ^ (typeString' ps) res ^ ")"
  | typeString' ps ANYTYPE = "<any type>"
```

```
val typeString = typeString' pathString

fun substString pairs =
  "{ " ^ String.concatWith ", " (map (fn (p, tau) => pathString p ^ " |--> " ^ typeString t

val tyexString : tyex -> string = typeString' (pathString' snd)
```

S532a. ⟨*definition of* `typeString` *for Molecule types* S531b⟩+≡ (S500b) ◁S531c S532b▷

```
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) =
  case c
    of COMPVAL tau => "[" ^ x ^ " : " ^ typeString tau ^ "]"
     | COMPABSTY _    => "(abstype " ^ x ^ ")"
     | COMPMANTY tau => "(type " ^ x ^ " " ^ typeString tau ^ ")"
     | COMPMOD mt => "(module [" ^ x ^ " : " ^ mtString mt ^ "])"

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 ^ ")"
```

Supporting code for Molecule

S532

S532b. ⟨*definition of* `typeString` *for Molecule types* S531b⟩+≡ (S500b) ◁S532a S532c▷

```
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) ^ ")"
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 ^ ")"
```

S532c. ⟨*definition of* `typeString` *for Molecule types* S531b⟩+≡ (S500b) ◁S532b

```
fun boolString b = if b then "#t" else "#f"
```

S532d. ⟨*definition of* `expString` *for Molecule* S532d⟩≡ (S500b) S533a▷

```
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
           | LETRECX (bs, e) => bracketSpace ["letrec", tbindings bs, expString e
           | LAMBDA (xs, body) => bracketSpace ("lambda" :: map formal xs @ [expSt
           | VCONX vcon => vconString vcon
           | CASE (e, matches) =>
               let fun matchString (pat, e) = sqbracket (spaceSep [patString pat,
               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
```

S533a. ⟨*definition of* `expString` *for Molecule* S532d⟩+≡ (S500b) ◁ S532d

```
  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
           | 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
```

S533b. ⟨*legacy test cases* S512b⟩+≡ ◁ S531a

```
  -> (val ah (mutable array int)$addh)
  <routine> : ((mutable array int) int -> )
  -> 1
  1 : int
```

```
-> (+ 3 3)
6 : int
-> int
type error: int names a type, but a variable is expected
-> 1
1 : int
-> 'hello
hello : sym
-> (int$+ 2 2)
4 : int
-> int$+
<routine> : (int int -> int)
-> (type A (mutable array int))
A = (mutable array int)
-> A$remh
<routine> : ((mutable array int) -> int)
-> A$addl
<routine> : ((mutable array int) int -> )
-> (var [arr : A])
arr : A
-> (var [test-int : int] [test-sym : sym] [test-null : null] [test-bool : bool])
test-int : int
test-sym : sym
test-null : null
test-bool : bool
-> arr
Run-time error: uninitialized variable arr
```

S534a. ⟨*result type of* K *should be* tau *but is* result S534a⟩≡ (S469b)

```
raise TypeError ("value constructor " ^ K ^ " should return " ^ typeString tau ^
                 ", but it returns type " ^ typeString result)
```

S534b. ⟨*type of* K *should be* tau *but is* tau' S534b⟩≡ (S469b)

```
raise TypeError ("value constructor " ^ K ^ " should have " ^ typeString tau ^
                 ", but it has type " ^ typeString tau')
```

T.12 Primitives

S534c. ⟨*primitives for Molecule* Int *module* :: S534c⟩≡ S535 ▷

```
("+", arithop op +, arithtype) ::
("-", arithop op -, arithtype) ::
("*", arithop op * , arithtype) ::
("/", arithop op div, arithtype) ::
```

We have two kinds of predicates: ordinary predicates take one argument, and comparisons take two. Some comparisons apply only to integers. (From here on, you can figure out the types for yourself—or get the ML compiler to tell you.) DUPLICATES ADT.

S534d. ⟨*primitives* [[**mcl**]] S492a⟩+≡ (S491b) ◁ S493

```
inject_bool  : bool -> value
project_bool : value -> bool
```

```
fun inject_bool x =
      CONVAL (PNAME (if x then "#t" else "#f"), [])
fun project_bool (CONVAL (PNAME "#t", [])) = true
  | project_bool (CONVAL (PNAME "#f", [])) = false
  | project_bool _ = raise RuntimeError "projected non-boolean"

fun inject_predicate f = fn x => inject_bool (f x)
fun predop f = unaryOp (inject_predicate f)
```

```
fun comparison f = binaryOp (inject_predicate f)
fun intcompare f = comparison (
                     fn (NUM n1, NUM n2) => f (n1, n2)
                      | _ => raise BugInTypeChecking "integers expected")
```

And here come the predicates. Equality comparison succeeds only on symbols and numbers. The empty list is dealt with through case expressions.

S535. ⟨*primitives for Molecule* `Int` *module* :: S534c⟩+≡ ◁ S534c

```
("<", intcompare op <, comptype inttype) ::
(">", intcompare op >, comptype inttype) ::
("=", intcompare op =, comptype inttype) ::
```

Chapter contents

U Supporting code for μSmalltalk

U.1 Implementations of some predefined classes

Classes whose implementations aren't shown in the chapter.

U.1.1 Methods of primitive classes

S537a. ⟨*methods of class* `Object` S537a⟩≡
```
(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))
```

S537b. ⟨*primitives for μSmalltalk* `::` S537b⟩≡ (S552a) S537e ▷
```
("sameObject", binaryPrim (mkBoolean o eqRep)) ::
("class",       classPrimitive) ::
("isKindOf",    binaryPrim kindOf) ::
("isMemberOf", binaryPrim memberOf) ::
("error",       binaryPrim error) ::
("subclassResponsibility",
               errorPrim "subclass failed to implement a method it was responsible for") ::
("leftAsExercise", errorPrim "method was meant to be implemented as an exercise") ::
```

S537c. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S537c⟩≡ (S548b) S550d ▷
```
fun errorPrim msg = fn _ => raise RuntimeError msg
```

S537d. ⟨*methods of class* `Class` S537d⟩≡
```
(method superclass () (primitive superclass self))
(method name () (primitive className self))
(method printProtocol () (primitive protocol self))
(method printLocalProtocol () (primitive localProtocol self))
(method compiledMethodAt: (aSymbol) (primitive getMethod self aSymbol))
(method addSelector:withMethod: (aSymbol aMethod) (primitive setMethod self aSymbol aMethod)
  self)
(method methodNames () (primitive methodNames self))
(method removeSelector: (aSymbol) (primitive removeMethod self aSymbol)
  self)
```

S537e. ⟨*primitives for μSmalltalk* `::` S537b⟩+≡ (S552a) ◁ S537b S538c ▷
```
("newUserObject", classPrim (fn (meta, c) => newUserObject c)) ::
("superclass", classPrim superclassObject) ::
("className", classPrim (fn (_, c) => mkSymbol (className c))) ::
("protocol", classPrim (protocols true)) ::
("localProtocol", classPrim (protocols false)) ::
```

```
("getMethod", binaryPrim getMethod) ::
("setMethod", setMethod o fst) ::
("removeMethod", binaryPrim removeMethod) ::
("methodNames", classPrim methodNames) ::
```

S538a. ⟨*methods of class* `UndefinedObject` S538a⟩≡

```
(method print () ('nil print) self)
```


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.

S538b. ⟨*predefined μSmalltalk classes and values* S538b⟩≡ S555e ▷

```
(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 S538d⟩
)
```

S538c. ⟨*primitives for μSmalltalk* :: S537b⟩+≡ (S552a) ◁ S537e S553b ▷

```
("value", valuePrim) ::
```

S538d. ⟨*tracing methods on class* `Block` S538d⟩≡ (S538b)

```
(method traceFor: (n) [locals answer]
    (set &trace n)
    (set answer (self value))
    (set &trace 0)
    answer)
(method trace () (self traceFor: -1))
```

S538e. ⟨*predefined μSmalltalk classes and values that use numeric literals* S538e⟩≡ (S560c) S539a ▷

```
(val &trace 0)
```

U.1.2 *Class* `Boolean`

S538f. ⟨*definition of class* `Boolean` S538f⟩≡

```
(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.1.3 Implementation of 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`.

S539a. ⟨*predefined μSmalltalk classes and values that use numeric literals* S538e⟩+≡ (S560c) ◁S538e S539b▷

```
(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.

S539b. ⟨*predefined μSmalltalk classes and values that use numeric literals* S538e⟩+≡ (S560c) ◁S539a

```
(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.1.4 Collection things

Class `Association`

valuePrim 699b

Method `associationsDo:` visits all the key-value pairs in a keyed collection. A key-value pair is represented by an object of class `Association`.

S539c. ⟨*collection classes* S539c⟩≡ (S560c) S540a▷

```
(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.

Implementation of `Dictionary`

A `Dictionary` is the simplest and least specialized of the keyed collections. If all µSmalltalk objects could be hashed, we would want to represent a `Dictionary` as a hash table. Because not every µSmalltalk object can be hashed, we 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`.

S540a. ⟨*collection classes* S539c⟩+≡ (S560c) ◁S539c S541c▷

```
(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 S540b⟩
)
```

The operations that `Dictionary` must implement are `associationsDo:`, `at:put`, and `removeKey:ifAbsent`. Iteration over associations can be delegated to the list of associations. To implement `at:put:`, we search for the association containing the given key. If we find such an association, we mutate its value. If we find no such association, we add one.

S540b. ⟨*other methods of class* `Dictionary` S540b⟩≡ (S540a) S540c▷

```
(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)
```

Removing a key requires that we first save the removed value, so we can answer it. The actual removal is done by sending the `reject:` message to the representation.

S540c. ⟨*other methods of class* `Dictionary` S540b⟩+≡ (S540a) ◁S540b S540d▷

```
(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:`.

S540d. ⟨*other methods of class* `Dictionary` S540b⟩+≡ (S540a) ◁S540c S541a▷

```
(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`.

S541a. ⟨*other methods of class* `Dictionary` S540b⟩+≡ (S540a) ◁S540d S541b▷

```
(method add: (anAssociation)
  (self at:put: (anAssociation key) (anAssociation value)))
```

A dictionary's `print` method uses `associationsDo:`.

S541b. ⟨*other methods of class* `Dictionary` S540b⟩+≡ (S540a) ◁S541a

```
(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)
```

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 S555f–S556b in Section U.2.3.

S541c. ⟨*collection classes* S539c⟩+≡ (S560c) ◁S540a S546▷

```
(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 670b⟩
)
```

Since it's not useful to create an array without specifying a size, I redefine class method `new` so that it 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 on page 728.

S541d. ⟨*other methods of class* `Array` **[[prototype]]** S541d⟩≡

```
(method select:  (_) (self error: 'select-on-arrays-left-as-exercise))
(method collect: (_) (self error: 'collect-on-arrays-left-as-exercise))
```

Like lists, arrays have keys from 0 to `size` − 1. I iterate over the keys.

U.1.5 Class `Number`, plus powers and roots

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.

S542a. ⟨*other methods of class* `Number` S542a⟩≡ (672a) S542b ▷

```
(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)))})}))
```


Our implementation of square root 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, we need 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] Our convergence condition is $|x_i - x_{i-1}| < \epsilon$. The default ϵ used in `sqrt` is $1/100$ Using `coerce:` ensures we can use the same `sqrt` method for both fractions and floats.

S542b. ⟨*other methods of class* `Number` S542a⟩+≡ (672a) ◁ S542a

```
(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>)
```

U.1.6 Implementation of integers

PERHAPS ALL WE REALLY NEED TO SEE HERE ARE THE THREE COERCIONS, PLUS TAKE NOTE OF `div:` AND `/`.

S542c. ⟨*other methods of class* `Integer` S542c⟩≡ (672c) S542d ▷

When integers are divided, the result isn't an integer; it's a fraction.

The integer method `timesRepeat:` executes a loop a finite number of times.

S542d. ⟨*other methods of class* `Integer` S542c⟩+≡ (672c) ◁ S542c

```
(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))}))
```

[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 $x_i \approx \sqrt{n}$.

The only concrete integer class built into μSmalltalk is `SmallInteger`. Almost all its methods are primitive. They are defined in chunks S554c–S555a.

S543a. ⟨*numeric classes* S543a⟩≡ (S560c) S543b ▷

```
(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 on page 731.

U.1.7 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 we can multiply two mantissas 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 arithmetic 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.

S543b. ⟨*numeric classes* S543a⟩+≡ (S560c) ◁ S543a

```
(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))}))
```

[2]Some implementations of ML reserve one bit as a dynamic-type tag or as a tag for the garbage collector.

```
        self)
    ⟨other methods of class Float S544a⟩
)
```

Like the other numeric classes, `Float` must provide methods that give a binary operation access to the representation of its argument.

S544a. ⟨*other methods of class* `Float` S544a⟩≡ (S543b) S544b ▷

```
(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. For this purpose, I add a private method `isZero`.

S544b. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁ S544a S544c ▷

```
(method < (x) ((self - x) isNegative))
(method = (x) ((self - x)   isZero))
(method isZero () (mant = 0))  ; private
```

Negation is easy: answer a new float with a negated mantissa.

S544c. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁ S544b S544d ▷

```
(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 35.

S544d. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁ S544c S544e ▷

```
(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.

S544e. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁ S544d S544f ▷

```
(method * (x-prime)
    (Float mant:exp: (mant * (x-prime mant)) (exp + (x-prime exp))))
```

We 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 we don't lose too much precision from m.

S544f. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁ S544e S545a ▷

```
(method reciprocal ()
    (Float mant:exp: (1000000000 div: mant) (-9 - exp)))
```

Coercing converts to `Float`, and converting `Float` to `Float` is the identity.

S545a. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁S544f S545b▷
```
(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.

S545b. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁S545a S545c▷
```
(method asInteger ()
   ((exp isNegative) ifTrue:ifFalse:
      {(mant div: (10 raisedToInteger: (exp negated)))}
      {(mant    * (10 raisedToInteger: exp))}))
```

To get a fraction, we either put a power of 10 in the denominator, or we make a product with 1 in the denominator.

S545c. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁S545b S545d▷
```
(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 three methods can be delegated to `Integer`.

S545d. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁S545c S545e▷
```
(method isNegative         () (mant isNegative))
(method isNonnegative      () (mant isNonnegative))
(method isStrictlyPositive () (mant isStrictlyPositive))
```

A floating-point number is printed as m`x10^`e. But we want to avoid printing a number like 77 as `770x10^-1`. So if the `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.

S545e. ⟨*other methods of class* `Float` S544a⟩+≡ (S543b) ◁S545d
```
(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.1.8 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: we need to know what is the abstraction, what is the representation, what is the abstraction function, what is the representation invariant, 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.

S546. ⟨*collection classes* S539c⟩+≡ (S560c) ◁S541c

```
(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) [locals looks-similar]
       (set looks-similar ((self size) = (s size)))
       (looks-similar ifTrue:
           {(self do: [block (x) ((s includes: x) ifFalse:
                                                {(set looks-similar false)})])})
       looks-similar)
)
```

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.

In addition to the methods shown in the class definition, class `Set` inherits `size`, `isEmpty`, `includes:`, `print`, and other methods from `Collection`.

U.2 INTERPRETER THINGS

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.

S547a. ⟨*abstract syntax and values for μSmalltalk* S547a⟩≡ (S547c)
```
⟨support for μSmalltalk stack frames S551b⟩
⟨definitions of exp, rep, and class for μSmalltalk 694a⟩
⟨definitions of value and method for μSmalltalk 693⟩
⟨definition of def for μSmalltalk 695b⟩
⟨definition of unit_test for μSmalltalk S547b⟩
⟨definition of xdef (shared) generated automatically⟩
fun className (CLASS {name, ...}) = name
⟨definition of valueString for μSmalltalk S567c⟩
⟨definition of expString for μSmalltalk S569d⟩
```

S547b. ⟨*definition of* `unit_test` *for μSmalltalk* S547b⟩≡ (S547a)
```
⟨definition of unit_test for untyped languages (shared) generated automatically⟩
              | CHECK_PRINT of exp * string
```

And overall structure...

The evaluator is built on top of everything else, and finally ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S559b⟩ reads the initial basis, then closes the cycles by calling the functions from ⟨*support for bootstrapping classes/values used during parsing* S551d⟩.

The code in the interpreter is organized so that the ⟨*support for bootstrapping classes/values used during parsing* S551d⟩ is as early as possible, immediately following the definition of ⟨*abstract syntax and values for μSmalltalk* S547a⟩ and the associated utility functions. Afterward come parsing, primitives, and evaluation. The code for ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S559b⟩ comes almost at the end, just before the execution of the command line. The full structure of the interpreter resembles the structure of the μScheme interpreter shown in chunk S373a, with the addition of chunks for bootstrapping and for stack tracing.

S547c. ⟨*usm.sml* S547c⟩≡
```
⟨shared: names, environments, strings, errors, printing, interaction, streams, & initialization S237a⟩

⟨abstract syntax and values for μSmalltalk S547a⟩
⟨support for logging (for coverage analysis) S548a⟩
⟨utility functions on μSmalltalk classes, methods, and values S549c⟩

⟨support for bootstrapping classes/values used during parsing S551d⟩

⟨lexical analysis and parsing for μSmalltalk, providing filexdefs and stringsxdefs S560e⟩

⟨evaluation, testing, and the read-eval-print loop for μSmalltalk S559a⟩

⟨implementations of μSmalltalk primitives and definition of initialBasis S559b⟩
⟨function runAs for μSmalltalk, which prints stack traces S568a⟩
⟨code that looks at command-line arguments and calls runAs to run the interpreter generated automatically⟩
⟨type assertions for μSmalltalk generated automatically⟩
```

CLASS	694c
type exp	696a

S548a. ⟨*support for logging (for coverage analysis)* S548a⟩≡ (S547c)
```
val logging =
  String.isSubstring ":log:" (":" ^ getOpt (OS.Process.getEnv "BPCOPTIONS", "") ^ ":")
fun q s = "\"" ^ s ^ "\""
val _ = if logging then println "val ops = ...\n" else ()

fun logSend srcloc msgname =
  app print [ "\nops.SEND { loc = ", q (srclocString srcloc)
            , ", selector = ", q msgname, " }\n" ]
fun logFind name candidate =
  app print ["\nops.findMethod { selector = ", q name
             , ", on = ", q (className candidate), "}\n"]

fun logClass name (ms : method list) =
  let fun subclassExp (SEND (_, _, "subclassResponsibility", _)) = true
        | subclassExp (BEGIN [e]) = subclassExp e
        | subclassExp _ = false
      val subclassM = subclassExp o #body
      val methodNames = commaSep o map (q o #name)
  in  app print [ "\nops.class { name = ", q name, ", methods = { " , methodNames ms
                , " }, subclass_responsibilities = { "
                , methodNames (List.filter subclassM ms), " } }\n"
                ]
  end

fun logGetMethod class m =
  app print ["\nops.getMethod { class = ", q class, ", method = ", q m, " }\n"]

fun logSetMethod class m =
  app print ["\nops.setMethod { class = ", q class, ", method = ", q m, " }\n"]
```

The interpreter has one more circularity to manage. Before we can define values of the built-in classes, we have to define the classes themselves. And before we can define the built-in classes, we have to define the primitives that are used in those classes. But there are primitives that depend on `nil`, which is a value of a built-in class! For example, when we create a new array, its contents are initially `nil`. To arrange for the right definitions to appear in the right order, I organize code for primitives and built-in classes in two layers.

The first layer includes chunks ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S537c⟩ and ⟨*built-in classes* `Object` *and* `UndefinedObject` generated automatically⟩. This code defines `Object`, which enables us to define `UndefinedObject`, which enables us to define `nilValue` (the internal representation of `nil`). The second layer includes chunks ⟨*ML code for remaining classes' primitives* S552d⟩ and ⟨*remaining built-in classes* generated automatically⟩. They define all the other primitives and built-in classes, some of which use `nilValue`.

S548b. ⟨*support for primitives and built-in classes* S548b⟩≡ (S559a)

⟨*utility functions for building primitives in μSmalltalk* S552b⟩
⟨*metaclass utilities* S550c⟩
⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S537c⟩
⟨*utility functions for parsing internal method definitions* S549a⟩
⟨*built-in class* `Object` 704a⟩
⟨*built-in class* `UndefinedObject` *and value* `nilValue` 704b⟩
⟨*ML code for remaining classes' primitives* S552d⟩
⟨*built-in classes* `Class` *and* `Metaclass` 704d⟩
⟨*metaclasses for built-in classes* 703c⟩

Order of definition:

```
(object undef nilValue class metaclass
 object-meta undef-meta class-meta meta-meta)
```

Utility functions for parsing internal method definitions

S549a. ⟨*utility functions for parsing internal method definitions* S549a⟩≡ (S548b)

```
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"
```


Utilities

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.

S549b. ⟨*helper functions for evaluation* S549b⟩≡ (S559a)

```
fun optimizedBind (x, v, xi) =
  let val loc = find (x, xi)
  in  (loc := v; xi)
  end handle NotFound _ => bind (x, ref v, xi)
```

S549c. ⟨*utility functions on* μ*Smalltalk classes, methods, and values* S549c⟩≡ (S547c) S549d ▷

```
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`. To get access to information about a class, we have to write a pattern match. When all we want is a class's name or its unique identifier, pattern matching is fairly heavy notation, so I provide 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.

S549d. ⟨*utility functions on* μ*Smalltalk classes, methods, and values* S549c⟩+≡ (S547c) ◁ S549c S549

```
className : class -> name
classId   : class -> metaclass ref
```

```
fun className (CLASS {name,  ...}) = name
fun classId   (CLASS {class, ...}) = class
```

We extract a method's name using another convenience function, `methodName`. Other manipulations of methods include `renameMethod`, which is used when a user class wants to use a primitive method with a name other than the one I built in, and `methods`, which builds an environment suitable for use in a class.

S549e. ⟨*utility functions on* μ*Smalltalk classes, methods, and values* S549c⟩+≡ (S547c) ◁ S549d S550a ▷

```
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
```

In general, I make a new class by calling mkClass, which checks to be sure that no instance variable is repeated. Each class is uniquely identified by its class field, which points to a unique mutable location.

S550a. ⟨*utility functions on µSmalltalk classes, methods, and values* S549c⟩+≡ (S547c) ◁ S549e

```
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 S550b⟩
  ; CLASS { name = name, super = SOME super, ivars = ivars
          , methods = ref (methodsEnv ms), class = ref meta }
  )
```

S550b. ⟨*if any name in* ivars *repeats a name declared in a superclass, raise* RuntimeError S550b⟩≡ (S550a)

```
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
```

S550c. ⟨*metaclass utilities* S550c⟩≡ (S548b)

```
fun setMeta (CLASS { class = m as ref PENDING, ... }, meta) = m := META meta
  | setMeta (CLASS { class = ref (META _), ... }, _) =
      raise InternalError "double patch"
```

- Value super is the superclass from which the new class inherits; superMeta is super's metaclass. Class super is bound into user-defined instance methods, and class superMeta is bound into user-defined class methods. These bindings guarantee that every message sent to SUPER arrives at the proper destination.
- Function method builds the representation of a method from its syntax.
- 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, I apply foldr to addMethodDefn, a pair of empty lists, and the list of method definitions ms.

S550d. ⟨*ML functions for* Object*'s and* UndefinedObject*'s primitives* S537c⟩+≡ (S548b) ◁ S537c S551a ▷

```
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'
        | addMethodDefn (m as { flavor = IMETHOD, ... }, (c's, i's)) = (c's, method m :: i'
  in  foldr addMethodDefn ([], []) ms
  end
```

The object named as a superclass must in fact represent a class, so its representation must be CLASSREP c, where c is the class it represents. That object is an instance of its metaclass. Function findClass returns the metaclass and the class.

S551a. ⟨*ML functions for* Object*'s and* UndefinedObject*'s primitives* S537c⟩+≡ (S548b) ◁S550d S553d▷

```
fun findClass (supername, xi) =     findClass : name * value ref env -> class * class
  case !(find (supername, xi))
    of (meta, CLASSREP c) => (meta, c)
     | v => raise RuntimeError ("object " ^ supername ^ " = " ^ valueString v ^
                                " is not a class")
```


U.2.1 Stack frames

S551b. ⟨*support for µSmalltalk stack frames* S551b⟩≡ (S547a)

```
datatype frame = FN of int
local
  val next_f = ref 0
in
  fun newFrame () = FN (!next_f) before next_f := !next_f + 1
end

type active_send = { method : name, class : name, loc : srcloc }

val noFrame = newFrame () (* top level, unit tests, etc... *)

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

fun raString (FN n) = "F@-" ^ intString n
```

S551c. ⟨*reraise* Return*, adding* msgname*,* class*, and* loc *to* unwound S551c⟩≡ (697b)

```
let val this = { method = msgname, class = className class, loc = srcloc }
in  raise Return { value = v, to = F', unwound = this :: unwound }
end
```

U.2.2 Bootstrapping

Blocks I use the technique again for blocks. I could actually get away without bootstrapping the Block class, but by defining Block and Boolean together, I clarify their relationship, especially the implementations of the whileTrue: and whileFalse: methods.

S551d. ⟨*support for bootstrapping classes/values used during parsing* S551d⟩≡ (S547c) S557a▷

```
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
```

CLASS	694c
type class	694c
class	697b
className	S549d
CLASSREP	694a
CLOSURE	694a
CMETHOD	695b
find	311b
findClass	706b
IMETHOD	695b
InternalError	S366f
intString	S238f
logClass	S548a
logging	S548a
member	S240b
META	694c
methodsEnv	S549e
msgname	697b
type name	310a
PENDING	694c
Return	695a
RuntimeError	S366c
srcloc	697b
unwound	697b
valueString	S567c

U.2.3 Primitives

To find a primitive by name, I look it up in the association list `primitives`.

S552a. ⟨*definition of* `primitives` S552a⟩≡ (S559a)

```
val primitives = ⟨primitives for μSmalltalk :: S537b⟩ nil
```

Utilities for creating primitives

Most primitives are created directly from ML functions. As in the interpreter for μScheme (Chapter 5), I build what I need in stages. Here I first turn unary and binary functions into primitives, then turn primitives into methods.

S552b. ⟨*utility functions for building primitives in μSmalltalk* S552b⟩≡ (S548b) S552c ▷

```
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) : primitive
fun binaryPrim f = (fn ([a, b], _) => f (a, b) | (vs, _) => arityError 1 vs) : primitive
```

A few primitives are more easily created as user methods. To make it easy to create user methods I define function `userMethod`. There is one dodgy bit: the `superclass` field of the user method. Because this class is used only to define the meaning of messages to `super`, and because none of my predefined user methods sends messages to `super`, I can get away with a bogus superclass that understands no messages. The bogus superclass is not the actual superclass of the class where the method will be used, but no program can tell the difference.

Function `internalExp` is an auxiliary function used to parse a string into abstract syntax; it calls parser `exp` from Section U.3.2.

S552c. ⟨*utility functions for building primitives in μSmalltalk* S552b⟩+≡ (S548b) ◁ S552b

```
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 class primitives take both the metaclass and the class as arguments.

S552d. ⟨*ML code for remaining classes' primitives* S552d⟩≡ (S548b) S552e ▷

```
classPrim : (class * class -> value) -> primitive
```

```
fun classPrim f =
  unaryPrim (fn (meta, CLASSREP c) => f (meta, c)
              | _ => raise RuntimeError "class primitive sent to non-class")
```

S552e. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁ S552d S553a ▷

```
fun superclassObject (_, CLASS { super = NONE, ... })   = nilValue
  | superclassObject (_, CLASS { super = SOME c, ... }) = classObject c
```

Arithmetic with overflow

The implementations of the primitives are easy; we try to build a block containing the result, but if the computation overflows, we answer the overflow block instead.

S553a. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁ S552e S554c ▷

```
fun withOverflow binop ([(_, NUM n), (_, NUM m), ovflw], xi) =
      (mkBlock ([], [VALUE (mkInteger (binop (n, m)))], emptyEnv, objectClass, noFrame)
       handle Overflow => ovflw)
  | withOverflow _ ([_, _, _], _) =
      raise RuntimeError "numeric primitive with overflow expects numbers"
  | withOverflow _ _ =
      raise RuntimeError "numeric primitive with overflow expects 3 arguments"
```


S553b. ⟨*primitives for μSmalltalk* :: S537b⟩+≡ (S552a) ◁ S538c S553c ▷

```
("addWithOverflow", withOverflow op + ) ::
("subWithOverflow", withOverflow op - ) ::
("mulWithOverflow", withOverflow op * ) ::
```

Hashing

S553c. ⟨*primitives for μSmalltalk* :: S537b⟩+≡ (S552a) ◁ S553b S555a ▷

```
("hash", unaryPrim (fn (_, SYM s) => mkInteger (fnvHash s)
                     | v => raise RuntimeError "hash primitive expects a symbol")) ::
```

Object primitives

Object identity My primitive method decides whether objects are identical by comparing their representations. Here's how I justify the cases:

- ML equality on arrays *is* object identity.
- Because numbers and symbols are immutable in both Smalltalk and ML, I can use ML equality on numbers and symbols, and it appears to the μSmalltalk programmer that I am using object identity.
- 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.

S553d. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S537c⟩+≡ (S548b) ◁ S551a S

`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
```

Printing By default, an object prints as its class name in angle brackets.

Class membership For `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.

S554a. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S537c⟩+≡ (S548b) ◁S553d S554b▷

```
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"
```

The `error:` primitive raises `RuntimeError`.

S554b. ⟨*ML functions for* `Object`*'s and* `UndefinedObject`*'s primitives* S537c⟩+≡ (S548b) ◁S554a

```
fun error (_, (_, SYM s)) = raise RuntimeError s
  | error (_, (c, _    )) =
      raise RuntimeError ("error: got class " ^ className c ^ "; expected Symbol")
```

Integer primitives

Integers print using the `intString` function defined in Appendix I.

S554c. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S553a S554d▷

```
fun printInt (self as (_, NUM n)) = ( xprint (intString n); self )
  | printInt _ = raise RuntimeError ("cannot print when object inherits from Int")
```

The also support UTF-8 printing.

S554d. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S554c S554e▷

```
fun printu (self as (_, NUM n)) = ( printUTF8 n; self )
  | printu _ = raise RuntimeError ("receiver of printu is not a small integer")
```

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.

S554e. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S554d S554f▷

```
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)
```

To create a new integer, you must pass the integer class, plus an argument that is represented by an integer.

S554f. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S554e S555b▷

```
fun newInteger ((_, CLASSREP c), (_, NUM n)) = (c, NUM n)
  | newInteger _ = raise RuntimeError ("made new integer with non-int or non-class")
```

Here are the primitive operations on small integers.

S555a. ⟨*primitives for μSmalltalk* :: S537b⟩+≡ (S552a) ◁S553c S555d▷
```
("newSmallInteger",   binaryPrim newInteger) ::
("+",    arithop op +  )  ::
("-",    arithop op -  )  ::
("*",    arithop op *  )  ::
("div", arithop op div)  ::
("<",    intcompare op <) ::
(">",    intcompare op >) ::
("printSmallInteger", unaryPrim printInt) ::
("printu",            unaryPrim printu)   ::
```
In chunk S543a, these primitives are used to define class `SmallInteger`.

Symbol primitives

A symbol prints as its name, with no leading `'`.

S555b. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S554f S555c▷
```
fun printSymbol (self as (_, SYM s)) = (xprint s; self)
  | printSymbol _ = raise RuntimeError "cannot print when object inherits from Symbol"
```
To create a new symbol, you must pass an argument that is represented by a symbol.

S555c. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S555b S555f▷
```
fun newSymbol ((_, CLASSREP c), (_, SYM s)) = (c, SYM s)
  | newSymbol _ = raise RuntimeError ("made new symbol with non-symbol or non-class")
```
S555d. ⟨*primitives for μSmalltalk* :: S537b⟩+≡ (S552a) ◁S555a S556b▷
```
("printSymbol", unaryPrim  printSymbol) ::
("newSymbol",   binaryPrim newSymbol   ) ::
```
There is no need to create `Symbol` internally, so we put it in the initial basis.

S555e. ⟨*predefined μSmalltalk classes and values* S538b⟩+≡ ◁S538b S557f▷
```
(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))
)
```
Array primitives

The primitive operations on arrays are creation, subscript, update, and size.

A new array contains all `nil`.

S555f. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S555c S555g▷
```
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"
```
To create primitives that expect `self` to be an array, we define `arrayPrimitive`.

S555g. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S555f S556a▷

`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"

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.

S556a. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S555g S556c▷

```
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
```


Here are all the primitive array methods.

S556b. ⟨*primitives for μSmalltalk* :: S537b⟩+≡ (S552a) ◁S555d

```
("arrayNew",    binaryPrim     newArray)    ::
("arraySize",   arrayPrimitive arraySize)   ::
("arrayAt",     arrayPrimitive arrayAt)     ::
("arrayUpdate", arrayPrimitive arrayUpdate) ::
```

In chunk S541c, these primitive methods are used to define class `Array`.

Block primitives

Class primitives

Showing protocols The `showProtocol` function helps implement the `protocol` and `localProtocol` primitives, which are defined on class `Class`. Its implementation is not very interesting. Function `insert` helps implement an insertion sort, which we use to present methods in alphabetical order.

S556c. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S556a S557b▷

```
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
```

S557a. ⟨*support for bootstrapping classes/values used during parsing* S551d⟩+≡ (S547c) ◁S551d

```
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
```

S557b. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S556c S557c▷

```
fun methodNames (_, CLASS { methods, ... }) = mkArray (map (mkSymbol o fst) (!methods))
```

S557c. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S557b S557d▷

```
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"
```

S557d. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S557c S557e▷

```
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"
```

S557e. ⟨*ML code for remaining classes' primitives* S552d⟩+≡ (S548b) ◁S557d

```
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 { name = _, formals = xs, locals = ys, body = e, superclass = _ }
          val m' = { name = s, formals = xs, locals = ys, body = e
                   , superclass = superclass }
          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] =
      raise RuntimeError ("setMethod primitive given non-method " ^ valueString
  | setMethod [(_, CLASSREP _), s, _] =
      raise RuntimeError ("setMethod primitive given non-symbol " ^ valueString
  | setMethod [c, _, _] =
      raise RuntimeError ("setMethod primitive given non-class " ^ valueString (
  | setMethod _ =
      raise RuntimeError "setMethod primitive given wrong number of arguments"
```

S557f. ⟨*predefined μSmalltalk classes and values* S538b⟩+≡ ◁S555e S560c▷

```
(class CompiledMethod
  [subclass-of Object]
)
```

U.2.4 Evaluation tracing

The trace function is given an action with which to perform the send; action is run by applying to the empty tuple. If tracing is enabled, trace emits two tracing messages: one before and one after running the action. The job of knowing whether tracing is enabled, and of emitting messages if so, is delegated to functions traceIndent and traceOutdent, each of which takes a tracing action of the form fn () => ..., which is executed only if tracing is enabled.

S558a. ⟨*definition of function* trace S558a⟩≡ (697b)

```
fun trace action =
  let val (file, line) = srcloc
      val () =
        traceIndent (msgname, (file, line)) xi (fn () =>
          let val c   = className startingClass
              val obj = if String.isPrefix "class " c then c
                        else "an object of class " ^ c
          in  [file, ", line ", intString line, ": ",
               "Sending message (", spaceSep (msgname :: map valueString vs), ")",
               " to ", obj]
          end)
      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
```

Functions traceIndent and outdentTrace, are defined in ⟨*exposed tracing functions* S566b⟩. This chunk also defines function eprintlnTrace, which is called from chunks S559c and S568a to show the stack of active message sends after a run-time error.

U.2.5 Evaluating extended definitions

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.

S558b. ⟨*evaluation,* basis*, and* processDef *for μSmalltalk* S558b⟩≡ (S559a)

```
type basis = value ref env
fun processDef (d, xi, interactivity) =
  let val (xi', v) = evaldef (d, xi)
      val _ = if prints 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.

Extended definitions are evaluated by the shared read-eval-print loop. And because of the way primitives are used in the evaluator, it needs more supporting code than in other bridge languages.

S559a. ⟨*evaluation, testing, and the read-eval-print loop for μSmalltalk* S559a⟩≡ (S547c)
 ⟨*shared definition of* `withHandlers` generated automatically⟩
 ⟨*support for primitives and built-in classes* S548b⟩
 ⟨*definition of* `newClassObject` *and supporting functions* 703a⟩
 ⟨*functions for managing and printing a μSmalltalk stack trace* S565b⟩
 ⟨*definition of* `primitives` S552a⟩
 ⟨*helper functions for evaluation* S549b⟩
 ⟨*definition of the* `Return` *exception* 695a⟩
 ⟨*evaluation,* `basis`*, and* `processDef` *for μSmalltalk* S558b⟩
 ⟨*shared unit-testing utilities* S246d⟩
 ⟨*definition of* `testIsGood` *for μSmalltalk* S568b⟩
 ⟨*shared definition of* `processTests` S247b⟩
 ⟨*shared read-eval-print loop and* `processPredefined` generated automatically⟩

U.2.6 *Initializing, bootstrapping, and running the interpreter*

The first entries in the initial basis are the primitive classes. Each one needs a metaclass to be an instance of. To be faithful to Smalltalk, the subclass relationships of the metaclasses should be isomorphic to the subclass relationships of the classes. This is true for user-defined classes created with `newClassObject`, but on the primitive classes, I cheat: the metaclasses for `UndefinedObject` and `Class` inherit directly from `Class`, not from `Object`'s metaclass.

S559b. ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S559b⟩≡ (S547c)

```
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 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.

S559c. ⟨*implementations of μSmalltalk primitives and definition of* `initialBasis` S559b⟩+≡ (S547

```
val initialXi =
  let val xdefs =
        stringsxdefs ("predefined classes",
                      ⟨predefined μSmalltalk classes and values, as strings (from chunk 664)⟩)
      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
```

Before we can close the cycles, we have to create VAL bindings for true and false. Because the parser prevents user code from binding true and false, we can't do this in μSmalltalk; the val bindings are written in ML.

S560a. ⟨*implementations of μSmalltalk primitives and definition of* initialBasis S559b⟩+≡ (S547c) ◁ S559c S560b ▷

```
local
  fun newInstance classname = SEND (nullsrc, VAR classname, "new", [])
in
  val initialXi = processPredefined (VAL ("true",  newInstance "True" ), initialXi)
  val initialXi = processPredefined (VAL ("false", newInstance "False"), initialXi)
end
```

Once we've read the class definitions, we can close the cycles, update the ref cells, and we're almost ready to go. By this time, all the necessary classes should be defined, so if any cycle fails to close, we halt the interpreter with a fatal error.

S560b. ⟨*implementations of μSmalltalk primitives and definition of* initialBasis S559b⟩+≡ (S547c) ◁ S560a S560d ▷

```
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 numeric and collection classes are in the initial basis.

S560c. ⟨*predefined μSmalltalk classes and values* S538b⟩+≡ ◁ S557f

⟨*numeric classes* S543a⟩
⟨*predefined μSmalltalk classes and values that use numeric literals* S538e⟩
⟨*collection classes* S539c⟩

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 written in ML.

S560d. ⟨*implementations of μSmalltalk primitives and definition of* initialBasis S559b⟩+≡ (S547c) ◁ S560b

```
val initialXi = processPredefined (VAL ("nil", VALUE nilValue), initialXi)
val initialBasis = initialXi
```

U.3 Lexing and parsing

S560e. ⟨*lexical analysis and parsing for μSmalltalk, providing* filexdefs *and* stringsxdefs S560e⟩≡ (S547c)

⟨*lexical analysis for μSmalltalk* S560f⟩
⟨*parsers for single μSmalltalk tokens* S562a⟩
⟨*parsers and parser builders for formal parameters and bindings* generated automatically⟩
⟨*parsers and* xdef *streams for μSmalltalk* S561d⟩
⟨*shared definitions of* filexdefs *and* stringsxdefs S254c⟩

U.3.1 Lexical analysis

There are two reasons we can't reuse μScheme's lexer for μSmalltalk: 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.

A source-code location includes a name for the source, plus line number.

S560f. ⟨*lexical analysis for μSmalltalk* S560f⟩≡ (S560e) S561a ▷

```
val nullsrc : srcloc = ("internally generated SEND node", 1)
```

The representation of a token is almost the same as in μScheme. The differences are that there are two kinds of brackets, and that a # character does not introduce a Boolean.

S561a. ⟨*lexical analysis for μSmalltalk* S560f⟩+≡ (S560e) ◁S560f S561b▷

```
datatype pretoken = INTCHARS of char list
                  | NAME     of name
                  | QUOTE    of string option (* symbol or array *)
type token = pretoken plus_brackets
```


To produce error messages, we must be able to convert a token back to a string.

S561b. ⟨*lexical analysis for μSmalltalk* S560f⟩+≡ (S560e) ◁S561a S561c▷

```
fun pretokenString (INTCHARS ds)     = implode ds
  | pretokenString (NAME    x)       = x
  | pretokenString (QUOTE NONE)      = "'"
  | pretokenString (QUOTE (SOME s)) = "'" ^ s
```

S561c. ⟨*lexical analysis for μSmalltalk* S560f⟩+≡ (S560e) ◁S561b

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.3.2 Parsing

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. To decide the issue, we use the *first* character of a message's name.

S561d. ⟨*parsers and* `xdef` *streams for μSmalltalk* S561d⟩≡ (S560e) S562b▷

```
fun arity name =
      let val cs = explode name
      in  if Char.isAlpha (hd cs) then
            length (List.filter (fn c => c = #":") cs)
          else
            1
      end

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  errorAt ("in " ^ what ^ ", message " ^ msgname ^ " expects " ^
                         argn (arity msgname) ^ ", but gets " ^
                         argn (length args)) loc
  end
```

Here's the parser.

S562a. ⟨*parsers for single μSmalltalk tokens* S562a⟩≡ (S560e)

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) <$>? token
val name = (fn (NAME s)         => SOME s  | _ => NONE) <$>? 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 any_name = name

val int = intFromChars <$>! intchars
```

Supporting code for μSmalltalk

S562

S562b. ⟨*parsers and* xdef *streams for μSmalltalk* S561d⟩+≡ (S560e) ◁S561d S562c▷

```
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
```

S562c. ⟨*parsers and* xdef *streams for μSmalltalk* S561d⟩+≡ (S560e) ◁S562b S562d▷

```
val atomicExp =  LITERAL <$> NUM    <$> int
             <|> LITERAL <$> SYM    <$> (sym <|> (quote *> name)
                                             <|> (quote *> (intString <$> int)))
             <|> SUPER              <$  eqx "super" name
             <|> VAR                <$> name
```

S562d. ⟨*parsers and* xdef *streams for μSmalltalk* S561d⟩+≡ (S560e) ◁S562c S563a▷

⟨*parsers and parser builders for formal parameters and bindings* generated automatically⟩

```
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 <*> 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")
  ]
```

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.

S563a. ⟨*parsers and* xdef *streams for μSmalltalk* S561d⟩+≡ (S560e) ◁S562d S563b▷

```
exp      : exp parser
quotelit : value parser
```

```
fun exp tokens = (
      atomicExp
  <|> quote       *> (VALUE <$> quotelit) (* not while reading predefined classe
  <|> curlyBracket ("{exp ...}", curry BLOCK [] <$> 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) = errorAt ("sent message " ^ m ^ " to no object") loc
and noMsg (loc, e) = errorAt ("found receiver " ^ expString e ^ " with no messag
and messageSend receiver (loc, msgname) args =
      if arityOk msgname args then
          OK (SEND (loc, receiver, msgname, args))
      else
          arityErrorAt loc "message send" msgname args
```

If any μSmalltalk code tries to change any of the predefined "pseudovariables," the settable parser causes an error.

The remaining parser functions are mostly straightforward. The quotelit function may call mkSymbol, mkInteger, or mkArray, which must not be called until after the initial basis is read in. Function quotelit is recursive and is called by exp, so I define it as if it were mutually recursive with exp.

S563b. ⟨*parsers and* xdef *streams for μSmalltalk* S561d⟩+≡ (S560e) ◁S563a S563c▷

```
quotelit : value parser
```

```
and 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
```

Function unit_test parses a unit test.

S563c. ⟨*parsers and* xdef *streams for μSmalltalk* S561d⟩+≡ (S560e) ◁S563b S563d▷

```
testtable : unit_test parser
```

```
val printable = name <|> implode <$> intchars

val testtable = usagePasers
  [ ("(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)
  ]
```

The parser for definitions 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.

S563d. ⟨*parsers and* xdef *streams for μSmalltalk* S561d⟩+≡ (S560e) ◁S563c S564a▷

```
method : method_def parser
```

```
val method =
  let fun method kind name impl =
```

```
            check (kname kind, 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, name, (formals as (loc, xs), locals, body)) =
            if arityOk name xs then
              OK (xs, locals, BEGIN body)
            else
              arityErrorAt loc (kind ^ " definition") name xs
      val mb = method_body exp
  in  usageParsers
      [ ("(method f (args) body)", method IMETHOD <$> name <*>! mb "method")
      , ("(class-method f (args) body)",
                                   method CMETHOD <$> name <*>! mb "class method")
      ]
  end
val parseMethods = many method <* many eol <* eos
```

True definitions.

S564a. ⟨*parsers and* `xdef` *streams for µSmalltalk* S561d⟩+≡ (S560e) ◁S563d S564b▷

`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.

S564b. ⟨*parsers and* `xdef` *streams for µSmalltalk* S561d⟩+≡ (S560e) ◁S564a S565a▷

```
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"
```

S565a. ⟨*parsers and* `xdef` *streams for μSmalltalk* S561d⟩+≡ (S560e) ◁S564b

```
val xdefstream = interactiveParsedStream (smalltalkToken, xdef)
```

U.4 Support for tracing

Tracing support is divided into three parts: support for printing indented messages, which is conditioned on the value of the variable `&trace`; support for maintaining a stack of source-code locations, which is used to provide information when an error occurs; and exposed tracing functions, which are used in the main part of the interpreter. To keep the details hidden from the rest of the interpreter, the first two parts are made `local`.

S565b. ⟨*functions for managing and printing a μSmalltalk stack trace* S565b⟩≡ (S559a)

```
local
  ⟨private state and functions for printing indented traces S565c⟩
  ⟨private state and functions for maintaining a stack of source-code locations S566a⟩
in
  ⟨exposed tracing functions S566b⟩
end
```

The `traceMe` function is used internally to decide whether to trace; it not only returns a Boolean but also decrements `&trace` if needed.

S565c. ⟨*private state and functions for printing indented traces* S565c⟩≡ (S565b) S565d▷

`traceMe : value ref env -> bool`

```
fun traceMe xi =
  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
```

The local variable `tindent` maintains the current trace state; `indent` uses it to print an indentation string.

S565d. ⟨*private state and functions for printing indented traces* S565c⟩+≡ (S565b) ◁S565c S565e▷

```
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`. For consistency, we outdent from the previous level *before* printing a message; we indent from the current level *after* printing a message.

S565e. ⟨*private state and functions for printing indented traces* S565c⟩+≡ (S565b) ◁S565d

```
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
        ()
```

Printing of trace messages is conditional, but we always maintain a stack of source-code locations. The stack is displayed when an error occurs.

S566a. ⟨*private state and functions for maintaining a stack of source-code locations* S566a⟩≡ (S565b)

```
val locationStack = ref [] : (string * srcloc) list ref
fun push srcloc = locationStack := srcloc :: !locationStack
fun pop () = case !locationStack
               of []     => raise InternalError "tracing stack underflows"
                | h :: t => locationStack := t
```

Here are the tracing-related functions that are exposed to the rest of the interpreter. The interpreter uses `traceIndent` to trace sends, `outdentTrace` to trace answers, and `resetTrace` to reset indentation. And it uses `eprintlnTrace` to print an error message, show the stack trace, and reset the trace.

S566b. ⟨*exposed tracing functions* S566b⟩≡ (S565b)

```
resetTrace     : unit -> unit
traceIndent    : string * srcloc -> value ref env -> (unit -> string list) -> unit
outdentTrace   :                    value ref env -> (unit -> string list) -> unit
showStackTrace : bool -> unit
eprintlnTrace  : string -> unit
```

```
fun resetTrace ()       = (locationStack := []; tindent := 0)
fun traceIndent what xi = (push what; tracePrint INDENT_AFTER   xi)
fun outdentTrace     xi = (pop ();    tracePrint OUTDENT_BEFORE xi)

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)

fun findRepeat xs k =
  if k > 20 then
    (0, [], xs)
  else
    let val repeat as (n, _, _) = removeRepeat k xs
    in  if n >= 3 then
          repeat
        else
          findRepeat xs (k + 1)
    end

fun findRepeatAfter xs 10 = ([], (0, [], xs))
  | findRepeatAfter xs  k =
      let val (n, header, ys) = findRepeat (List.drop (xs, k)) 1
      in  if n > 0 then
            (List.take(xs, k), (n, header, ys))
          else
            findRepeatAfter xs (k + 1)
```

```
      end handle Subscript => ([], (0, [], xs))

  fun showStackTrace condense =
    if null (!locationStack) then
      ()
    else
      let fun show (msg, (file, n)) =
            app xprint ["  Sent '", msg, "' in ", file, ", line ", intString n, "\n"]
          val preRepeat =
            if condense then findRepeatAfter (!locationStack) 0
            else ([], (0, [], !locationStack))
          val _ = xprint "Method-stack traceback:\n"
      in  case preRepeat
            of ([], (0, _, locs)) => app show locs
             | (_,  (0, _, _)) => let exception Invariant in raise Invariant end
             | (prefix, (k, header, locs)) =>
                 ( app show prefix
                 ; if null prefix then ()
                   else app xprint [ "    ... loop of size "
                                   , Int.toString (length header) , " begins ...\n"
                                   ]
                 ; app show header
                 ; app xprint [ "    ... loop of size ", Int.toString (length header)
                              , " repeated ", Int.toString k, " times ...\n"
                              ]
                 ; app show locs
                 )
      end
  fun eprintlnTrace s = ( eprintln s
                        ; showStackTrace (String.isSubstring "recursion too deep" s
                                          orelse String.isSubstring "CPU time exha
                        ; resetTrace ()
                        )
```


S567a. ⟨*report* (return e) *escapes* frames S567a⟩≡

```
  if null frames then
    raise RuntimeError
      ("tried to (return " ^ expString e ^ ") from an activation that has died")
  else
    raise RuntimeError
      ("tried to (return " ^ expString e ^ ") from an activation that has died " ^
       "[stack trace would have " ^ countString frames "frame" ^ "]")
```

S567b. ⟨*report* return *escapes* frames S567b⟩≡ (701c)

```
  if null frames 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  " ^
                        separate ("", "\n  ") (map activeSendString frames))
```

To avoid confusion, tracing code typically avoids print methods; instead, it uses valueString to give information about a value.

S567c. ⟨*definition of* valueString *for μSmalltalk* S567c⟩≡ (S547a)

```
  fun valueString (c, NUM n) = intString n ^ valueString(c, USER [])
    | valueString (_, SYM v) = v
    | valueString (c, _) = "<" ^ className c ^ ">"
```

To trace method calls, μSmalltalk uses a custom runAs function; instead of eprintln, it calls eprintlnTrace.

activeSendString	S551b
className	S547a
eprintln	S238a
frames	701c
INDENT_AFTER	S565e
InternalError	S366f
intString	S238f
NUM	694a
OUTDENT_BEFORE	S565e
RuntimeError	S366c
separate	S239a
SYM	694a
tindent	S565d
tracePrint	S565e
USER	694a
xprint	S238b

S568a. ⟨*function* `runAs` *for μSmalltalk, which prints stack traces* S568a⟩≡ (S547c)

`runAs : interactivity -> unit`

```
fun runAs interactivity =
  let val _ = setup_error_format interactivity
      val prompts = if prompts interactivity then stdPrompts else noPrompts
      val xdefs = filexdefs ("standard input", TextIO.stdIn, prompts)
  in  ignore (readEvalPrintWith eprintlnTrace (xdefs, initialBasis, interactivity))
  end
```

U.5 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: testing for equality requires a call to `eval`, and if something is wrong with a value, we can't convert the value to a string—all we can do with a value is print it.

S568b. ⟨*definition of* `testIsGood` *for μSmalltalk* S568b⟩≡ (S559a)

```
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 testSimilar (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 S568c⟩
      ⟨definition of checkPrintPasses S569c⟩
      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
```

This thing is not like the others, because printing values *must* go to standard output.

S568c. ⟨*definitions of* `check{Expect,Assert,Error{Passes` *that call* `printsAs` S568c⟩≡ (S568b) S568d ▷

```
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]
```

S568d. ⟨*definitions of* `check{Expect,Assert,Error{Passes` *that call* `printsAs` S568c⟩+≡ (S568b) ◁ S568c S569

```
val cxfailed = "check-expect failed: "
fun checkExpectPasses (checkx, expectx) =
  case (outcome checkx, outcome expectx)
    of (OK check, OK expect) =>
         (case withHandlers (OK o testSimilar) (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 similarity 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]
```

S569a. ⟨*definitions of* `check{Expect,Assert,Error}Passes` *that call* `printsAs` S568c⟩+≡ (S568b) ◁S568d S569b▷

```
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]
```

S569b. ⟨*definitions of* `check{Expect,Assert,Error}Passes` *that call* `printsAs` S568c⟩+≡ (S568b) ◁S569a

```
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]
```

S569c. ⟨*definition of* `checkPrintPasses` S569c⟩≡ (S568b)

```
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]
```

S569d. ⟨*definition of* `expString` *for μSmalltalk* S569d⟩≡ (S547a)

```
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
```

V Supporting code for μProlog

This Appendix is longer than many others:

- Even Prolog's simple syntax requires more code to parse than prefix-parenthesized syntax.
- In μProlog, as in C, a comment can span multiple lines, which means its lexical analyzer has to track source-code locations. This tracking needs extra code.
- A μProlog interpreter has two modes: rule mode and query mode. Tracking modes introduces additional complexity.

V.1 Substitution

A substitution θ is a structure-preserving mapping from terms to terms. As in Chapter 7, we represent a substitution as an environment. The environment maps logical variables to terms. All the substitution functions resemble the functions used to substitute types for type varibles in Chapter 7.

S571a. ⟨*substitutions for μProlog* S571a⟩≡ (S82b) S571b ▷

```
type subst
idsubst : subst
```

```
type subst = term env
val idsubst = emptyEnv
```

S571b. ⟨*substitutions for μProlog* S571a⟩+≡ (S82b) ◁ S571a S571c ▷

```
varsubst : subst -> (name -> term)
```

```
fun varsubst theta =
  (fn x => find (x, theta) handle NotFound _ => VAR x)
```

S571c. ⟨*substitutions for μProlog* S571a⟩+≡ (S82b) ◁ S571b S571d ▷

```
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
```

Given the ability to substitute in a term, we may also want to substitute in goals and clauses.

S571d. ⟨*substitutions for μProlog* S571a⟩+≡ (S82b) ◁ S571c S572a ▷

```
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 we can substitute in constraints.

S572a. ⟨*substitutions for μProlog* S571a⟩+≡ (S82b) ◁S571d S572b▷

```
fun consubst theta =                     consubst : subst -> (con -> con)
  let fun subst (t1 ~  t2) = termsubst theta t1 ~ termsubst theta t2
        | subst (c1 /\ c2) = subst c1 /\ subst c2
        | subst TRIVIAL    = TRIVIAL
  in  subst
  end
```


We create substitutions using the same infix operator as in Chapter 7.

S572b. ⟨*substitutions for μProlog* S571a⟩+≡ (S82b) ◁S572a S572c▷

```
infix 7 |-->                                   |--> : name * term -> subst
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)
```

Substitutions compose just as in Chapter 7.

S572c. ⟨*substitutions for μProlog* S571a⟩+≡ (S82b) ◁S572b

```
fun dom theta = map (fn (a, _) => a) theta     compose : subst * subst -> subst
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.2 Unit testing

Unit testing in μProlog is different from any other unit testing: we check for satisfiability, or when given an explicit substitution, we check that the substitution satisfies the given query.

S572d. ⟨*definition of* `testIsGood` *for μProlog* S572d⟩≡ (S87b)

```
fun testIsGood (test, database) =          testIsGood : unit_test * basis -> bool
  let ⟨definitions of checkSatisfiedPasses and checkUnsatisfiablePasses S572e⟩
      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, we print it using function `qstring`.

S572e. ⟨*definitions of* `checkSatisfiedPasses` *and* `checkUnsatisfiablePasses` S572e⟩≡ (S572d) S572f▷

```
type query = goal list                        type query
val qstring = separate ("?", ", ") o map goalString    qstring : query -> string
```

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`.

S572f. ⟨*definitions of* `checkSatisfiedPasses` *and* `checkUnsatisfiablePasses` S572e⟩+≡ (S572d) ◁S572e S573

```
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.

S573a. ⟨*definitions of* `checkSatisfiedPasses` *and* `checkUnsatisfiablePasses` S572e⟩+≡ (S572d) ◁ S572f S573b ▷

```
  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.

S573b. ⟨*definitions of* `checkSatisfiedPasses` *and* `checkUnsatisfiablePasses` S572e⟩+≡ (S572d) ◁ S573a

```
  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.3 String conversions

This code converts terms, goals, and clauses to strings.

S573c. ⟨*definitions of* `termString`, `goalString`, *and* `clauseString` S573c⟩≡ (S58f) S574a ▷

```
  fun termString (APPLY ("cons", [car, cdr])) =
        let fun tail (APPLY ("cons", [car, cdr])) = ", " ^ termString car ^ tail cdr
              | 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) [")"] t)
```

S574a. ⟨*definitions of* `termString`, `goalString`, *and* `clauseString` S573c⟩+≡ (S58f) ◁S573c S574b▷

```
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)
```

S574b. ⟨*definitions of* `termString`, `goalString`, *and* `clauseString` S573c⟩+≡ (S58f) ◁S574a

```
fun substString pairs =
      separate ("no substitution", ", ")
      (map (fn (x, t) => x ^ " = " ^ termString t) pairs)
```

V.4 Lexical analysis

S574c. ⟨*lexical analysis and parsing for μProlog, providing* `cqstream` S574c⟩≡ (S87a)

```
⟨lexical analysis for μProlog S574d⟩
⟨parsers and streams for μProlog S577b⟩
val xdefstream = xdefsInMode RMODE
⟨shared definitions of filexdefs and stringsxdefs S254c⟩
```

V.4.1 Tokens

μProlog has a more complex lexical structure than other languages. We have uppercase, lowercase, and symbolic tokens, as well as integers. It simplifies the parser if we distinguish reserved words and symbols using `RESERVED`. Finally, because a C-style μProlog comment can span multiple lines, we have to be prepared for the lexical analyzer to encounter end-of-file. Reading 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`.

S574d. ⟨*lexical analysis for μProlog* S574d⟩≡ (S574c) S574e▷

`type token`

```
datatype token
  = UPPER     of string
  | LOWER     of string
  | SYMBOLIC  of string
  | INT_TOKEN of int
  | RESERVED  of string
  | EOF
```

We need to print tokens in error messages.

S574e. ⟨*lexical analysis for μProlog* S574d⟩+≡ (S574c) ◁S574d S575b▷

```
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.4.2 Classification of characters

The other languages in this book treat only parentheses, digits, and semicolons specially. But in Prolog, we distinguish two kinds of names: symbolic and alphanumeric. A symbolic name like + is used differently from an alphanumeric name like `add1`. This difference is founded on a different classification of characters.

In μProlog, every character is either a symbol, an alphanumeric, a space, or a delimiter.

S575a. ⟨*character-classification functions for μProlog* S575a⟩≡ (S575d)

```
val symbols = explode "!%^&*-+:=|~<>/?`$\\"
fun isSymbol c = List.exists (fn c' => c' = c) symbols
fun isIdent  c = Char.isAlphaNum c orelse c = #"_"
fun isDelim  c = not (isIdent c orelse isSymbol c)
```


V.4.3 Reserved words and anonymous variables

Tokens formed from symbols or from lower-case letters are usually symbolic, but sometimes they are reserved words. And because the cut is nullary, not binary, it is treated as an ordinary symbol, just like any other nullary predicate.

S575b. ⟨*lexical analysis for μProlog* S574d⟩+≡ (S574c) ◁S574e S575c▷

```
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.

S575c. ⟨*lexical analysis for μProlog* S574d⟩+≡ (S574c) ◁S575b S575d▷

```
fun anonymousVar () =
  case freshVar ""
    of VAR v => UPPER v
     | _ => let exception ThisCan'tHappen in raise ThisCan'tHappen end
```

V.4.4 Converting characters to tokens

We consume a stream of characters, intersperse with `EOL` (end-of-line) markers. We must product a stream of tokens. And unlike our other lexers, the μProlog lexer must produce *located* tokens, i.e., tokens that are tagged with source-code locations. The location corresponding to the start of the character stream is passed as a parameter to `tokenAt`.

S575d. ⟨*lexical analysis for μProlog* S574d⟩+≡ (S574c) ◁S575c

```
local
  ⟨character-classification functions for μProlog S575a⟩
  ⟨lexical utility functions for μProlog S575e⟩
in
  ⟨lexical analyzers for for μProlog S576c⟩
end
```

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.

S575e. ⟨*lexical utility functions for μProlog* S575e⟩≡ (S575d) S576a▷

```
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 ^ "'")
```


Utility function `unrecognized` is called when the lexical analyzer cannot recognize a sequence of characters. If the sequence is empty, it means there's no token. If anything else happens, an error has occurred.

S576a. ⟨*lexical utility functions for μProlog* S575e⟩+≡ (S575d) ◁ S575e S576b ▷

```
unrecognized : char list error -> ('a error * 'a error stream) option
```

```
fun unrecognized (ERROR _) = let exception Can'tHappen in raise Can'tHappen end
  | unrecognized (OK cs) =
      case cs
        of []        => NONE
         | #";" :: _ => let exception Can'tHappen in raise Can'tHappen end
         | _ =>
             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.

S576b. ⟨*lexical utility functions for μProlog* S575e⟩+≡ (S575d) ◁ S576a

```
fun nextline (file, line) = (file, line+1)
```

```
nextline : srcloc -> srcloc
```

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.

S576c. ⟨*lexical analyzers for for μProlog* S576c⟩≡ (S575d) S576d ▷

```
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.

S576d. ⟨*lexical analyzers for for μProlog* S576c⟩+≡ (S575d) ◁ S576c S576e ▷

```
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 Char.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.

S576e. ⟨*lexical analyzers for for μProlog* S576c⟩+≡ (S575d) ◁ S576d S577a ▷

```
ordinaryToken : token prolog_lexer
```

```
val ordinaryToken =
      underscore            <$> eqx #"_" char <*>! manySat isIdent
  <|> (RESERVED o str)      <$> sat isDelim char
  <|> int                   <$> intChars    <*>! manySat isIdent
```

```
  <|> (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
```

We need 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.

S577a. ⟨*lexical analyzers for for μProlog* S576c⟩+≡ (S575d) ◁S576e


```
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.5 Parsing

V.5.1 Utilities for parsing μProlog

S577b. ⟨*parsers and streams for μProlog* S577b⟩≡ (S574c) S577c ▷

```
symbol : string parser
upper  : string parser
lower  : string parser
int    : int    parser
```

```
type 'a parser = (token, 'a) polyparser
val token = token : token parser (* make it monomorphic *)
val symbol = (fn SYMBOLIC  s => SOME s | _ => NONE) <$>? token
val upper  = (fn UPPER     s => SOME s | _ => NONE) <$>? token
val lower  = (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)
```

We use these combinators to define the grammar from Figure D.2. We use `notSymbol` to ensure that a term like `3 + X` is not followed by another symbol. This means we don't parse such terms as `3 + X + Y`.

S577c. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S577b S578a ▷

```
notSymbol : unit parser
```

```
val notSymbol =
  symbol <!> "arithmetic expressions must be parenthesized" <|>
  pure ()
```

Parser nilt uses the empty list of tokens to represent the empty list of terms. It needs an explicit type constraint to avoid falling afoul of the value restriction on polymorphism. Function cons combines two terms, which is useful for parsing lists.

S578a. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S577c S578b▷

```
fun nilt tokens = pure (APPLY ("nil", [])) tokens
fun cons (x, xs) = APPLY ("cons", [x, xs])
```

```
nilt : term parser
cons : term * term -> term
```

Here is one utility function commas, plus renamings of three other functions.

S578b. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S578a S578c▷

```
variable        : string parser
binaryPredicate : string parser
functr          : string parser
commas : 'a parser -> 'a list parser
```

```
val variable        = upper
val binaryPredicate = symbol
val functr          = lower
fun commas p =
  curry op :: <$> p <*> many (reserved "," *> p)
```

I spell "functor" without the "o" because in Standard ML, functor is a reserved word.

V.5.2 *Parsing terms, atoms, and goals*

We're now ready to parse μProlog. The grammar is based on the grammar from Figure D.2 on page S55, except that I'm using named function to parse atoms, and I use some specialized tricks to organize the grammar. Concrete syntax is not for the faint of heart.

S578c. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S578b S579a▷

```
term   : term parser
atom   : term parser
commas : 'a parser -> 'a list parser
```

```
fun closing bracket = reserved bracket <?> bracket
fun wrap left right p = reserved left *> p <* closing right
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 element")
                                           <|> nilt
                                            )
          <|> nilt
```

```
                )
          )
          tokens
    end
```

Terms and goals shared 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.

S579a. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S578c S579b▷


```
asGoal : srcloc -> term -> goal error
goal   : goal parser
```

```
  fun asGoal _   (APPLY g) = OK g
    | asGoal loc (VAR v)   =
        errorAt ("Variable " ^ v ^ " cannot be a predicate") loc
    | asGoal loc (LITERAL n) =
        errorAt ("Integer " ^ intString n ^ " cannot be a predicate") loc

  val goal = asGoal <$> srcloc <*>! term
```

V.5.3 *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 I parse them first into a value of type `concrete`, and I worry about the interpretation later.

S579b. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S579a S579c▷

```
type concrete
```

```
  datatype concrete
    = BRACKET of string
    | CLAUSE  of goal * goal list option
    | GOALS   of goal list
    | CTEST   of unit_test
```

Among the unit tests, parsing `check-satisfied` is a bit tricky: we get 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.

S579c. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S579b S580a▷

```
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.

S580a. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S579c S580b▷

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, ...])")
```

Compared with unit tests, `concrete` values are easy to parse.

S580b. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S580a S580c▷

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 cases, we know what a `concrete` value is supposed to mean, but there's one case in which we don't: a phrase like "`color(yellow).`" could be either a clause *or* a query. To know which is meant, we have 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.

S580c. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S580b S580d▷

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 S58d in Chapter 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:

S580d. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S580c S580e▷

type xdef_or_mode

```
datatype xdef_or_mode
  = XDEF of xdef
  | NEW_MODE of mode
```

The next level of μProlog's parser interpreters a `concrete` value according to the mode. `BRACKET` values and unite 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.

S580e. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S580d S581a▷

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.

S581a. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S580e S581b▷

```
                                   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.5.4 *Reading clauses and queries while tracking locations and modes*

To produce a stream of definitions, every other language in this book uses the function `interactiveParsedStream` from page S280b. μProlog can't: `interactiveParsedStream` doesn't tag tokens with locations, and it doesn't keep track of modes. As a replacement, I define a somewhat more complex function, `cqstream`, below. At the core of `cqstream` is function `getXdef`.

S581b. ⟨*parsers and streams for μProlog* S577b⟩+≡ (S574c) ◁S581a

```
            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 cqstream S582a⟩

        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 J expect a stream of tokens tagged with `INLINE`. Even though we don't need `INLINE` for μProlog, it is easier to use a meaningless `INLINE` than it is to rewrite big chunks of Appendix J.

Function `getXdef` uses `startsWithEOF` to check if the input stream has no more tokens.

S582a. ⟨*utility functions for* `cqstream` S582a⟩≡ (S581b) S582b ▷

`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.

S582b. ⟨*utility functions for* `cqstream` S582a⟩+≡ (S581b) ◁ S582a S582c ▷

`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
```

Function `getXdef` tracks the prompt, the mode, and the remaining unread tokens, which together form the `read_state`. It also, when called, sets the prompt.

S582c. ⟨*utility functions for* `cqstream` S582a⟩+≡ (S581b) ◁ S582b

`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 S583a⟩
  )
```


Parser `xdef_or_mode` is always supposed to return something. If it doesn't, I issue an epic error message.

S583a. ⟨*fail epically with a diagnostic about* `tokens` S583a⟩≡ (S582c)

```
let exception ThisCan'tHappenCqParserFailed
    val tokensStrings =
      map (fn t => " " ^ tokenString t) o valOf o peek (many token)
    val _ = app print (tokensStrings tokens)
in  raise ThisCan'tHappenCqParserFailed
end
```

V.6 Command line

μProlog'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.

S583b. ⟨*function* `runAs` *for* μ*Prolog* S583b⟩≡ (S87a)

`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.

S583c. ⟨*code that looks at* μ*Prolog's command-line arguments and calls* `runAs` S583c⟩≡ (S87a)

```
fun runmain ["-q"]           = runAs (NOT_PROMPTING, PRINTING)
  | runmain []               = runAs (PROMPTING,     PRINTING)
  | runmain ("-trace" :: t) = (tracer := app eprint; runmain t)
  | runmain _   =
      TextIO.output (TextIO.stdErr,
                     "Usage: " ^ CommandLine.name() ^ " [trace] [-q]\n")
val _ = runmain (CommandLine.arguments())
```

Tracing code is helpful for debugging.

S583d. ⟨*support for tracing* μ*Prolog computation* S583d⟩≡ (S87a)

```
val tracer = ref (app print)
val _ = tracer := (fn _ => ())
fun trace l = !tracer l
```

V *Supporting code for μProlog*

IX. Code index

Code index