Chapter 3

Scheme, S-expressions, and first-class functions

Contents

3.1 Overview of μScheme and this chapter 63
3.2 Language I: S-expressions 65
3.3 Practice I: Recursive functions on lists 68
3.3.1 Basics 68
3.3.2 List reversal and the method of accumulating parameters 70
3.3.3 Prime numbers 71
3.3.4 Sorting 72
3.3.5 Sets 72
3.3.6 Association lists 74
3.4 Interlude: Laws of applicative programming 75
3.4.1 List laws 75
3.4.2 Classifying operations 76
3.4.3 Boolean laws 77
3.4.4 More functions, more laws 77
3.4.5 Proving properties of programs 77
3.5 Language II: Local variables and `let` 82
3.6 Practice II: Recursive functions on binary trees 83
3.7 Language III: First-class functions 87
3.7.1 Closures 90
3.7.2 Simple, useful higher-order functions 94
3.8 Practice III: Higher-order functions on lists 96
3.8.1 Standard higher-order functions on lists 96
3.8.2 Visualizing the standard list functions 98
3.8.3 Implementing the standard list functions 99
3.9 Practice IV: Higher-order functions for polymorphism 100
3.9.1 Approaches to polymorphism in Scheme 103
3.9.2 A polymorphic, higher-order sort 105
3.10 Practice V: Continuation-passing style 106
3.10.1 Continuation-passing for backtracking 108

3.11 Syntax and values of μScheme 113
3.11.1 Concrete syntax 113
3.11.2 Abstract syntax 114
3.11.3 Values 114
3.12 Operational semantics 115
3.12.1 Variables and functions 115
3.12.2 Rules for other expressions 119
3.12.3 Rules for evaluating definitions 121
3.13 The initial basis 123
3.14 The interpreter 125
3.14.1 Interfaces 125
3.14.2 Implementation of the evaluator 128
3.14.3 Evaluating definitions 133
3.14.4 Implementations of the primitives 135
3.14.5 Implementation of the interpreter's `main` procedure 140
3.14.6 Implementation of memory allocation 141
3.15 Large example: A metacircular evaluator 141
3.15.1 The environment and value store 142
3.15.2 Representations of values 142
3.15.3 The initial environment and store 143
3.15.4 The evaluator 144
3.15.5 Evaluating definitions 146
3.15.6 The `read-eval-print` loop 146
3.15.7 Tests 147
3.16 Scheme as it really is 147
3.16.1 Language differences 148
3.16.2 Proper tail calls 148
3.16.3 Data types 149
3.16.4 Macros 150
3.16.5 `call/cc` 150
3.17 Further reading 151
3.18 Exercises 152

Many programmers admire Lisp for its combination of power and simplicity. Inspired by ideas from mathematical logic, particularly by the *λ-calculus*, John McCarthy developed Lisp as a language for symbolic computation, which was called "list processing." McCarthy received ACM's Turing Award in 1971. For the kinds of computations typical of artificial intelligence, Lisp offers concise and natural programs, often close to mathematical definitions of the functions being computed. It has been used for most major AI programs for fifty years.

There have been many dialects of Lisp, but today's Lisp programmer probably uses one of two dialects: Common Lisp or Scheme. Common Lisp was designed to unify many of the dialects in use in the 1980s; its large, rich programming environment has often attracted large software projects. Scheme was designed to be small, clean, and powerful; its power and simplicity have attracted many teachers and authors, including me. A third Lisp dialect, Racket, has been designed especially for teaching; it actually consists of five nested dialects, the smallest of which is a language intended for beginning students, and the largest of which is intended for professional programmers (Felleisen et al. 2004).

Scheme was first developed by Guy Steele and Gerry Sussman, who introduced the main ideas in a classic series of MIT technical reports in the late 1970s, all bearing titles of the form "LAMBDA: The Ultimate —." A few years later, Scheme's protean ability to express many different programming-language ideas and to build programs in many different styles was shown off beautifully in a comprehensive textbook by Abelson and Sussman (1985).

So what is Scheme? The heart of a language may often be found not so much in the *constructs* it has but in the *values* it computes with. C would not be C without pointers and pointer arithmetic. Perl would not be Perl without strings and regular expressions. Fortran would not be Fortran without arrays. Scheme would not be Scheme without *lists* and *functions*.[1]

Scheme takes lists from original Lisp. Lists can contain other lists, so they can be used to build trees. Lists of key-value pairs can act as tables. Add numbers and symbols, and you have everything you need for symbolic computation.

Scheme also provides first-class, higher-order, nested functions. Because of a subtle difference in the meanings of nested functions, they are far more useful in Scheme than in original Lisp. One new language construct (`lambda`) and a handful of primitives lead to a dramatically different way of thinking about programming and computation.

As noted in Chapter 1, we work not with Scheme itself, but with μScheme (pronounced "micro-Scheme"), a distillation of Scheme's most essential features. I use the word "Scheme" to refer to ideas that Scheme and μScheme share. "Full Scheme" and "μScheme" refer to the large and small languages, respectively. This chapter includes a formal definition and an implementation of μScheme.

3.1 Overview of μScheme and this chapter

In Impcore, we have already seen three of the central ideas of Scheme, namely:

- *Interactive programming with functions.* Instead of "writing a program," Schemers "define a function;" instead of "running a program," they "evaluate an expression," often by applying a function to some arguments.

[1] If we consider Scheme as it is today, any list of its essential aspects whould also include hygienic macros. But hygienic macros were developed relatively late, in the 1980s and 1990s, well after the other foundations of Scheme were laid down. And sadly, macros are beyond the scope of this book.

- *Simple, regular syntax.* In particular, there is no infix syntax and therefore no operator precedence.
- *Recursion as the central control structure.* In combination with "S-expressions" (next section), the ubiquitous use of recursion is the most important innovation of Lisp.[2]

μScheme changes and extends Impcore in four significant ways.

- Functions are no longer distinct from values. Functions are *first-class*, which means they can be used in the same ways as any other value. For example, functions can be passed as arguments to other functions and returned as values from other functions; they can also be assigned to variables and even stored in lists.

  Functions that accept functions as arguments or return functions as results are called *higher-order functions*; functions that accept and return only non-functional values are called *first-order functions.* Impcore has only first-order functions.
- μScheme supports many more values, including not only integers and functions but general S-expressions.
- A new language construct, `lambda`, makes it easy to define anonymous, nested functions.
- A new family of constructs, the `let` family, makes it easy to define and use local variables.

These changes have two significant consequences for programmers.

- Much as arrays, loops, and assignment statements lead to an imperative style of programming, S-expressions, recursive functions, and `let` binding lend themselves to a new, *applicative* style of programming. As shown in Section 3.4.5, we can prove facts about applicative programs using simple algebra, without having to resort to metatheory.
- The ability to define first-class, *nested* functions leads to a powerful new programming technique: *higher-order programming.*

There are also significant implications for the semantics.

- In Impcore, environments map names to values, and we implement `set` by binding new values. In μScheme, environments map names to mutable *locations*, which contain values. We implement `set` by changing the contents of locations. Sections 3.7.1 and 3.12.1 present the subtleties.

This chapter is large, and it is broken into many sections. A road map may help.

- We begin by exploring applicative programming with recursive functions and lists, which are the basic building blocks of Scheme programs (Section 3.3).

[2]Interestingly, although recursion has been used by mathematicians for centuries for defining functions, computer scientists did not at first recognize recursion as a *practical* method. Alan Perlis said that when some of the world's leading computer scientists met in 1960 to design the language Algol-60, "We really did not understand the implications of recursion, or its value, ... McCarthy did, but the rest of us didn't." (Wexelblat 1981, p. 160).

- I present algebraic laws for some of Scheme's most important primitive functions, and we see how to use those laws to prove simple facts about Scheme functions (Section 3.4).
- I introduce `let` binding and local variables (Section 3.5), which we use in an example program for traversing trees (Section 3.6).
- First-class functions are the heart of Scheme and the heart of the chapter. After introducing the basic language construct, `lambda`, and its meaning (Section 3.7), I show you applications to list processing (Section 3.8), to code reuse via *polymorphism* (Section 3.9), and to backtracking search via *continuation passing* (Section 3.10).
- With these examples as context, we examine the syntax and semantics of μScheme (Sections 3.11 and 3.12), then the initial basis (Section 3.13) and an interpreter (Section 3.14).
- We wrap up the chapter with a larger example (a metacircular evaluator, in Section 3.15) and a discussion of full Scheme (Section 3.16).

3.2 Language I: S-expressions

In precisely the same sense that the values of Impcore are integers, the values of Scheme[3] are *S-expressions*, which is short for "symbolic expressions." Scheme gets much of its special power and appeal from the marvelous match between the recursive structure of S-expressions and the recursive definition of functions.

In μScheme, an *S-expression* is either a *symbol*, a *number*, a *Boolean*, a *procedure*, or a *list* $(S_1 \ldots S_n)$ of zero or more S-expressions. The list containing no elements, (), is called the empty list, or sometimes *nil*. μScheme uses machine integers as numbers, and it uses `#t` and `#f` as the Boolean values true and false. Lists are straightforward, but symbols are a bit harder to explain.

A symbol is a value that is a name. To paraphrase Kelsey, Clinger, and Rees (1998), the utility of symbols depends on the fact that two symbols are identical if and only if their names are spelled in the same way. This behavior is exactly the behavior of the `Name` type from the Impcore interpreter, and it is a property that is very useful for representing identifiers in programs, which is why not only the Impcore interpreter but also most implementations of Scheme use symbols for this purpose. Symbols are useful for many other purposes as well; for example, they are often used the way enumeration literals are used in Pascal or C.

μScheme comes with a much richer set of primitive operations than Impcore.

`car`: If S is the list $(S_1 ... S_n)$, where $n > 0$, then `(car` S`)` is S_1; if S is *nil*, it is erroneous to apply `car` to it.

`cdr`: If S is the list $(S_1 \cdots S_n)$, where $n > 0$, then `(cdr` S`)` is $(S_2 \cdots S_n)$; if S is *nil*, it is erroneous to apply `cdr` to it. If $n = 1$, `(cdr` S`)` is *nil*. The word "`cdr`" is pronounced as the word "could" followed by an R.

`cons`: If S is the list $(S_1 \cdots S_n)$, then `(cons` S' S`)` is the list $(S'\ S_1 \cdots S_n)$. If S is not a list, `(cons` S' S`)` is erroneous.[4]

[3] In the definition of Scheme as well as in the literature, values are often called "objects."

[4] This explanation is an oversimplification; the whole truth about `cons` is found in Section 3.4.1.

=: (= S_1 S_2) returns `#t` if S_1 and S_2 are both the same number, both the same symbol, both the same Boolean, or both *nil*; it returns `#f` otherwise. (To compare non-empty lists, use the non-primitive procedure `equal?`, which is shown in chunk 69a.)

`null?`, `boolean?`, `number?`, `symbol?`, `procedure?`: These return `#t` if their argument is of the indicated type, `#f` otherwise.

+: If S_1 and S_2 are numbers, (+ S_1 S_2) is their sum; otherwise (+ S_1 S_2) is erroneous.

-, *, /: These arithmetic operators are like +.

<, >: Both arguments must be numbers or the result is erroneous. These primitives return `#t` if the indicated condition holds; they return `#f` otherwise.

The function name `cons` stands for "construct," which makes sense, but the names `car` and `cdr` stand for, respectively, "contents of the address register" and "contents of the decrement register," which makes sense only if we are thinking about the machine-language implementation of Lisp on the IBM 704. The Racket dialect of Lisp uses the more sensible names `first` and `rest` (Felleisen et al. 2001).

The primitives above have no side effects. For example, applying a function like `car` to a list does not change the list. As in Impcore, `print` is the only primitive that has a side effect; all other side effects are implemented by explicit `set` expressions.

We would be ready to look at some μScheme function definitions except for one thing: we have not yet seen how to write S-expression *literals* in μScheme programs. They cannot simply be written in programs as they are on paper, because, for example, the list `(a b)` would be interpreted as the *application* of a function `a` to an argument `b`, rather than the list containing symbols `a` and `b`. To distinguish an S-expression literal from an expression that is to be evaluated, Scheme precedes the literal by a single quote (`'`). For example, the symbol `a` is written `'a`, and the list `(a b 3)` is written `'(a b 3)`. The most common S-expression literal is `'()`, for *nil*. Integer and Boolean literals may be written directly, without the single quote.

A bit more formally, the syntax of literals is:

literal ::= *integer* | `#t` | `#f` | `'`*S-exp*
S-exp ::= *literal* | *symbol* | (⟨*S-exp*⟩)

You may want to begin writing programs in μScheme by imitating the code in this and later sections. Scheme programmers rarely use assignment or iteration; in typical Scheme style, we do almost everything with recursion. You can also use functions defined in Impcore, such as `gcd`, which still work in μScheme.

`cdr` *P*
`cons` *P*
`null?` *P*

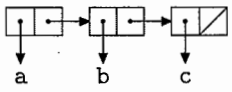
To understand basic list operations, consider these examples:

⟨*transcript* 66⟩≡ 68c▷

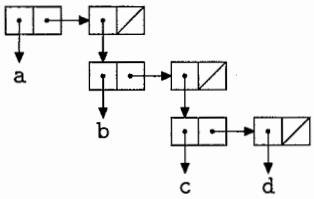
```
-> (cons 'a '())
(a)
-> (cons 'a '(b))
(a b)
-> (cons '(a) '(b))
((a) b)
-> (cdr '(a (b (c d))))
((b (c d)))
-> (null? '())
#t
-> (null? '(()))
#f
```

These examples illustrate the definitions of the primitives. The definitions must be applied carefully, which may not be as easy as it looks. For instance, `(a (b (c d)))` is a list of two elements, the symbol `a` and the list `(b (c d))`. Its `cdr`, therefore, is a list of one element, the just-named list, and is written `((b (c d)))`.

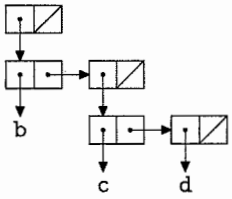
This behavior may be easier to understand if you picture any non-*nil* list as a box—called a *cons cell*—with two pointers, one to its `car` and the other to its `cdr`; if its `cdr` is *nil*, the second box is filled with a slash. For example, picture the list `(a b c)` as:



Using this graphical notation, we find the `car` or `cdr` of a nonempty list by following the arrow that leaves the left or right box of the first cons cell. We `cons` x and y by creating a new cons cell with pointers to x and y. As a more complicated example, consider the S-expression `(a (b (c d)))`:



Its `cdr` is simply what the first cell's right arrow points at, which is, as above, `((b (c d)))`:



Because all complex data structures are built from cons cells, old-school Scheme programmers frequently find themselves applying `car` or `cdr` several times in succession. These kinds of computations have traditional abbreviations:

67 ⟨*additions to the μScheme initial basis* 67⟩≡ 68a▷
```
(define caar (xs) (car (car xs)))
(define cadr (xs) (car (cdr xs)))
(define cdar (xs) (cdr (car xs)))
```
car P
cdr P

The chunk ⟨*additions to the μScheme initial basis* 67⟩ defines functions that are built into the μScheme interpreter itself, not as primitives, but as functions that are defined in terms of primitives.

Implementations of full Scheme must provide not only `caar` and `cadr`, but all functions of this form from `caar`, `cadr`, and `cdar` up to and including `cddddr`. Half of these functions are also in the initial basis of μScheme.

Another set of simple but useful functions are those that put their arguments into lists:

68a ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁67 68b▷

```
(define list1 (x)     (cons x '()))
(define list2 (x y)   (cons x (list1 y)))
(define list3 (x y z) (cons x (list2 y z)))
```

Full Scheme provides a single, *variadic* function, `list`, which puts any number of arguments into a list; see exercise 47.

3.3 Practice I: Recursive functions on lists

Having looked at the list primitives, we proceed to more interesting functions, which use these primitives. Much Scheme programming involves recursive functions on lists.

3.3.1 Basics

In Scheme, a list can be created in exactly two ways: by using `'()` or `cons`. Like any other data type that can be created in multiple ways, lists are normally consumed by functions that begin with case analysis. A function that consumes a list does its case analysis using `null?`, and in the case where the list is not null—it was created by using `cons`—a typical function makes a recursive call on the rest of the list.

One of the simplest such functions finds the length of a list.

68b ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁68a 69a▷

```
(define length (xs)
  (if (null? xs) 0
    (+ 1 (length (cdr xs)))))
```

The `length` function is typical of recursive functions that *consume* lists: the empty list is the *base case* where the recursion stops; the recursive call, which operates on `(cdr xs)`, is an *induction step.* The name `xs` (pronounced "exes") is intended to suggest a list of zero or more unknown elements, where a single `x` stands for an unknown element.

To help you understand how μScheme functions are evaluated, we examine how `length` behaves in excruciating detail. Consider this application:

68c ⟨*transcript* 66⟩+≡ ◁66 69b▷

```
-> (length '(a b))
2
```

We trace the application by considering the various calls to the `length` function and the value of parameter `xs` at each call.

cdr $\mathcal{P}$
cons $\mathcal{P}$
null? $\mathcal{P}$

- In the initial call, `xs` = `(a b)`.
- The list `(a b)` is not *nil*, so `(null? xs)` returns `#f`.

- The expression `(+ 1 (length (cdr xs)))` is evaluated, where `xs` = `(a b)`. The subexpression `(cdr xs)` evaluates to the list `(b)`. Let us follow the application of `length` to `(b)`:
  - List `xs` = `(b)`.
  - List `(b)` is not *nil*, so `(null? xs)` returns `#f`.
  - Expression `(+ 1 (length (cdr xs)))` is evaluated. Subexpression `(cdr xs)` is *nil*. We follow the application of `length` to *nil*:
    * `(null? xs)` returns `#t`.
    * `length` returns 0.
  - The call `(length (cdr xs))` returns 0, so `length` returns 1.
- The call `(length (cdr xs))` returns 1, so `length` returns 2.

Another basic list operation is to see if two lists are equal. We consider two lists equal if they have equal elements in corresponding positions. The primitive `=` tests for equality only of *atoms*, i.e., numbers, symbols, Booleans, and *nil*. We treat atoms as the base case, for which we use `=`; for the induction step, two lists are equal if their `car`s and `cdr`s are equal:

69a ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁68b 70a▷

```
⟨definitions of and, or, and not, for the initial basis 123a⟩
(define atom? (x) (or (number? x) (or (symbol? x) (or (boolean? x) (null? x)))))
(define equal? (s1 s2)
  (if (or (atom? s1) (atom? s2))
    (= s1 s2)
    (and (equal? (car s1) (car s2)) (equal? (cdr s1) (cdr s2)))))
```

69b ⟨*transcript* 66⟩+≡ ◁68c 70b▷

```
-> (equal? 'a 'b)
#f
-> (equal? '(a (1 3) c) '(a (1 3) c))
#t
-> (equal? '(a (1 3) d) '(a (1 3) c))
#f
-> (equal? #f #f)
#t
```

and	123a
boolean?	$\mathcal{P}$
car	$\mathcal{P}$
cdr	$\mathcal{P}$
null?	$\mathcal{P}$
number?	$\mathcal{P}$
or	123a
symbol?	$\mathcal{P}$

The `equal?` function shows a different pattern of recursion than the pattern used by `length`. That's because `equal?` consumes different data:

- The `equal?` function expects two arguments, not one.
- Each argument to the `equal?` function is an S-expression, not a list.
- An S-expression has a more complex structure than a list: for purposes of `equal?`, an S-expression is either an atom or is a list of S-expressions.

In `length`, `xs` is a list, and `(cdr xs)` is a list, but `(car xs)` is not a list. We can therefore expect to see recursive calls on `(cdr xs)` but never on `(car xs)`. In `equal?`, by contrast, both `s1` and `s2` are S-expressions, and if `s1` is not an atom, then *both* `(car s1)` and `(cdr s1)` are S-expressions, and similarly for `s2`. In `equal?`, we can therefore expect to see recursive calls on `(car s1)`, `(cdr s1)`, `(car s2)`, and `(cdr s2)`—as in fact we do.

Here is a third kind of example: like `equal?`, the `append` function consumes two arguments, but like `length`, the `append` function expects arguments that are lists, not general S-expressions. To append two lists `xs` and `ys`, we write a recursive `append` function that consumes only `xs`, leaving `ys` uninspected. The `append` function uses smaller and smaller values of `xs` on each recursive call. If `xs` is empty, then appending `ys` to `xs` simply produces `ys`.

70a ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁69a 70d▷

```
(define append (xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

70b ⟨*transcript* 66⟩+≡ ◁69b 70c▷

```
-> (append '(moon over) '(miami vice))
(moon over miami vice)
```

3.3.2 List reversal and the method of accumulating parameters

The reverse of the empty list is also the empty list. To reverse a nonempty list, we can use a mathematical law about list reversal. If R stands for reversal, x stands for one element in a list, and $\overline{X}$ stands for a list of any length, then

$$R(x\overline{X}) = R(\overline{X})R(x) = R(\overline{X})x.$$

To turn this law into code, we observe that any list to be reversed must be passed to a function. We'll call the function's formal parameter `xs`, so $x\overline{X}$ = `xs`. We can then conclude that when considered as a single element, x = `(car xs)`, and when considered as a list of length 1, x = `(list1 (car xs))`. We also conclude that $\overline{X}$ = `(cdr xs)`. Finally, we observe that to implement $R(\overline{X})x$, which is pronounced "reverse of big $\overline{X}$ followed by x," we need to consider x as a list and to implement "followed by" using `append`. A list-reversal function must also account for the possibility that `xs` is empty. The complete definition is as follows:

70c ⟨*transcript* 66⟩+≡ ◁70b 71b▷

```
-> (define simple-reverse (xs)
     (if (null? xs)
         xs
         (append (simple-reverse (cdr xs)) (list1 (car xs)))))
-> (simple-reverse '(my bonny lies over))
(over lies bonny my)
-> (simple-reverse '(a b (c d) e))
(e (c d) b a)
```

car 𝒫
cdr 𝒫
cons 𝒫
list1 125b
null? 𝒫

This `simple-reverse` function is expensive: `append` takes $O(n)$ time and space, and so `simple-reverse` takes $O(n^2)$ time and space, where n is the length of the list. We can do better by exploiting a different law: $R(x\overline{X})\overline{Y} = R(\overline{X})R(x)\overline{Y} = R(\overline{X})(x\overline{Y})$. The trick is to write a function `revapp` with *two* parameters; `xs` holds $x\overline{X}$, `ys` holds $\overline{Y}$, and `(revapp xs ys)` returns $R(\overline{X})\overline{Y}$: the reverse of `xs`, with `ys` appended.

70d ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁70a 71a▷

```
(define revapp (xs ys)
  (if (null? xs)
      ys
      (revapp (cdr xs) (cons (car xs) ys))))
```

This definition is an application of a technique called the *method of accumulating parameters*: the parameter `ys` is used to accumulate the eventual result. A compiler for full Scheme should translate this code into a very tight loop.

In the initial basis of μScheme, we define an *efficient* `reverse` by reversing and appending to the empty list.

71a ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁70d 74a▷

```
(define reverse (xs) (revapp xs '()))
```

71b ⟨*transcript* 66⟩+≡ ◁70c 71c▷

```
-> (reverse '(the atlantic ocean))
(ocean atlantic the)
```

3.3.3 Prime numbers

Here is a Scheme version of a well-known algorithm for finding prime numbers. The algorithm starts with sequence of numbers from 2 to n, and it produces a prime p by taking the first number in the sequence, then continuing recursively after removing from the sequence all multiples of p. Because it identifies a multiple of p by trying to divide by p, the algorithm is called *trial division*.[5]

We start with basic functions; `(seq m n)` returns the list `(m m+1 m+2 ... n)`, and `divides` tests if its first argument divides its second.

71c ⟨*transcript* 66⟩+≡ ◁71b 71d▷

```
-> (define seq (m n)
     (if (> m n) '() (cons m (seq (+ 1 m) n))))
-> (seq 3 7)
(3 4 5 6 7)
-> (define divides (p n) (= (mod n p) 0))
```

Calling `(remove-multiples p ns)` returns the list of numbers `ns` with all multiples of `p` removed.

71d ⟨*transcript* 66⟩+≡ ◁71c 71e▷

```
-> (define remove-multiples (p ns)
     (if (null? ns)
       '()
       (if (divides p (car ns))
         (remove-multiples p (cdr ns))
         (cons (car ns) (remove-multiples p (cdr ns))))))
-> (remove-multiples 2 '(2 3 4 5 6 7))
(3 5 7)
```

To find all the primes in a given sequence `ns`, we remove multiples repeatedly. If `ns` is empty, we find nothing. If `ns` is nonempty, we take the first element as a prime, remove its multiples from the remaining elements of the sequence, and recursively find more primes.

71e ⟨*transcript* 66⟩+≡ ◁71d 72a▷

```
-> (define find-primes-in (ns)
     (if (null? ns)
       '()
       (cons (car ns) (find-primes-in (remove-multiples (car ns) (cdr ns))))))
```

car	$\mathcal{P}$
cdr	$\mathcal{P}$
cons	$\mathcal{P}$
mod	123d
null?	$\mathcal{P}$
revapp	70d

[5] This algorithm is sometimes called the Sieve of Eratosthenes, but don't be fooled: O'Neill (2009) will convince you that this algorithm is not what Eratosthenes had in mind.

It is not sensible to pass just any sequence to `find-primes-in`. We must pass a sequence of numbers that begins with a prime p and that contains no multiples of primes smaller than p. To ensure these conditions are satisfied, and to bound the size of `ns`, we define the function `primes<=`, which finds all the primes up to a given `n`.

72a ⟨*transcript* 66⟩+≡ ◁71e 72b▷

```
-> (define primes<= (n) (find-primes-in (seq 2 n)))
-> (primes<= 10)
(2 3 5 7)
-> (primes<= 50)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
```

3.3.4 Sorting

The next example is a Scheme version of the sorting algorithm *insertion sort*. This sort has a very simple recursive structure: given a list of n elements, sort the last $n-1$ elements, then insert the first in its proper position. For the base case, where $n = 0$, the empty list is already sorted.

To translate this idea into code, let's first observe that insertion sort consumes a list. If the list is not empty, the obvious thing to do is recursively sort the `cdr`. But what are we to do with the `car`? We can't simply `cons` the `car` onto the front, as in `find-primes-in` above. Nor can we append it to the back, as we did in `simple-reverse` above. Instead, we have to insert the `car` into exactly the right place in the sorted `cdr`. The insertion is itself a nontrivial computation that consumes a list, so we should expect insertion to require a recursive function as well. It is actually the more difficult of the two functions:

72b ⟨*transcript* 66⟩+≡ ◁72a 72c▷

```
-> (define insert (x xs)
     (if (null? xs)
       (list1 x)
       (if (< x (car xs))
         (cons x xs)
         (cons (car xs) (insert x (cdr xs))))))
```

Given `insert`, it is easy to come up with the recursive `insertion-sort`:

72c ⟨*transcript* 66⟩+≡ ◁72b 73a▷

```
-> (define insertion-sort (xs)
     (if (null? xs)
       '()
       (insert (car xs) (insertion-sort (cdr xs)))))
-> (insertion-sort '(4 3 2 6 8 5))
(2 3 4 5 6 8)
```

car 𝒫
cdr 𝒫
cons 𝒫
find-primes-in 71e
list1 125b
null? 𝒫
seq 71c

3.3.5 Sets

In Scheme, as in any other language that supports lists, we can represent a set as a list with no repeated elements. Operations on this representation can be expensive, but the operations are easy to write and to understand. For example, adding a new element is like `cons`, provided the element is not already in the list.

Here is code for `emptyset`, `member?`, `add-element`, `size`, and `union` operations on sets. Only `member?` and `union` require explicit recursion. Functions `add-element` and `size` also consume lists, so they might be implemented recursively, but instead delegate the recursive parts of the problem to `member?` and `length` respectively. This example foreshadows a key idea in functional programming, which is developed at length in Section 3.7 on page 87: in a functional language, sometimes patterns of recursion can be reused.

73a ⟨*transcript* 66⟩+≡ ◁72c 73b▷

```
-> (val emptyset '())
-> (define member? (x s)
     (if (null? s)
       #f
       (if (equal? x (car s))
         #t
         (member? x (cdr s)))))
-> (define add-element (x s)
     (if (member? x s)
       s
       (cons x s)))
-> (define size (s)
     (length s))
-> (define union (s1 s2)
     (if (null? s1)
       s2
       (add-element (car s1) (union (cdr s1) s2))))
-> (val s (add-element 3 (add-element 'a emptyset)))
(3 a)
-> (member? 'a s)
#t
-> (union s (add-element 2 (add-element 3 emptyset)))
(a 2 3)
```

Using `equal?` in `member?` enables us to build sets of lists:

73b ⟨*transcript* 66⟩+≡ ◁73a 74c▷

```
-> (val t (add-element '(a b) (add-element 1 emptyset)))
((a b) 1)
-> (member? '(a b) t)
#t
```

If we had used = instead of `equal?`, this code would not have worked.

Other useful operations on sets appear at the end of the chapter, in Exercise 3 on page 156.

car 𝒫
cdr 𝒫
cons 𝒫
equal? 69a
length 68b
null? 𝒫

3.3.6 Association lists

One strength of Scheme is in dealing with symbolic information, and one of the essential data types for such information is the *table*. Small tables are often represented as *association lists*, or *a-lists*. An assocation list has the form $((k_1\ a_1)\ \ldots\ (k_m\ a_m))$, where the k_i are symbols (called *keys*) and the a_i are *attributes*. To make it easier to read and write code involving association lists, we define these auxiliary functions:

74a ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁71a 74b▷
```
(define mk-alist-pair (k a) (list2 k a))
(define alist-pair-key       (pair)  (car  pair))
(define alist-pair-attribute (pair)  (cadr pair))
(define alist-first-key       (alist) (alist-pair-key       (car alist)))
(define alist-first-attribute (alist) (alist-pair-attribute (car alist)))
```

The basic functions on association lists are `bind` and `find`, which add bindings to and retrieve attributes from association lists.

74b ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁74a 94a▷
```
(define bind (k a alist)
  (if (null? alist)
    (list1 (mk-alist-pair k a))
    (if (equal? k (alist-first-key alist))
      (cons (mk-alist-pair k a) (cdr alist))
      (cons (car alist) (bind k a (cdr alist))))))
(define find (k alist)
  (if (null? alist) '()
    (if (equal? k (alist-first-key alist))
      (alist-first-attribute alist)
      (find k (cdr alist)))))
```

74c ⟨*transcript* 66⟩+≡ ◁73b 74d▷
```
-> (val al (bind 'I 'Ching '()))
((I Ching))
-> (val al (bind 'E 'coli al))
((I Ching) (E coli))
-> (val al (bind 'I 'Magnin al))
((I Magnin) (E coli))
-> (find 'I al)
Magnin
```


cadr 67
car P
cdr P
cons P
equal? 69a
list1 68a
list2 68a
null? P

Association lists are somewhat like the environments used in the interpreter; indeed, in Section 3.15 we use association lists to *represent* environments.

More generally, one may want to associate a variety of *named* attributes with a given key. We can use an a-list in which the attribute of any key is itself an a-list. We call such a structure a *property list*. For example, a property list for fruits might be:

74d ⟨*transcript* 66⟩+≡ ◁74c 75a▷
```
-> (val fruits
     '((apple  ((texture crunchy) (color green)))
       (banana ((texture mushy)   (color yellow)))))
```

Functions to retrieve and add properties to a property list use the nested a-list structure:

75a ⟨*transcript* 66⟩+≡ ◁74d 75b▷

```
-> (define property (k p plist)
     (find p (find k plist)))
-> (define set-property (k p a plist)
     (bind k (bind p a (find k plist)) plist))
-> (property 'apple 'texture fruits)
crunchy
-> (set fruits (set-property 'apple 'color 'red fruits))
-> (property 'apple 'color fruits)
red
```

Another problem is to find all names that have a given value for a given property, e.g., all yellow things.

75b ⟨*transcript* 66⟩+≡ ◁75a 82a▷

```
-> (define has-property (p a alist) (= (find p alist) a))
-> (define gather-property (p a plist)
     (if (null? plist)
       '()
       (if (has-property p a (alist-first-attribute plist))
         (cons (alist-first-key plist) (gather-property p a (cdr plist)))
         (gather-property p a (cdr plist)))))
-> (set fruits (set-property 'lemon 'color 'yellow fruits))
-> (gather-property 'color 'yellow fruits)
(banana lemon)
```

3.4 Interlude: Laws of applicative programming

The examples from the preceding sections are *applicative*, i.e., they work by applying functions and creating new bindings, not by assigning values to variables. One advantage of programming without assignment is that we can use *algebraic laws* to simplify our code. Algebraic laws, such as the familiar $x + 0 = x$ or the more esoteric $\sin^2 x + \cos^2 x = 1$, state facts that we know about functions. In programming languages, such laws are used in two ways: to *specify behavior* of data types, and to *simplify code.*

alist-first-attribute 74a
alist-first-key 74a
bind 74b
cdr 𝒫
cons 𝒫
find 74b
fruits 74d
null? 𝒫

3.4.1 List laws

Most of what we need to know about the primitive functions `cons`, `car`, and `cdr` can be expressed by the following laws:

$$\texttt{(car (cons x y))} = \texttt{x}$$
$$\texttt{(cdr (cons x y))} = \texttt{y}$$

These laws reveal the whole truth about `cons`, namely, that it is legal to apply `cons` to *any* two arguments, even if the second argument is not a list. The secret of `cons` is that it constructs a pair; the use of pairs and *nil* to represent lists is merely a programming convention. This convention is built into Scheme's initial basis, including the `print` function, but it remains a convention, not a property of the language itself.

Having revealed the truth about `cons`, it is appropriate for us to introduce the primitive `pair?`, which returns `#t` if and only if its argument is a pair created with `cons`. The rest of what we need to know about `cons` and `pair?` is given by laws of this flavor:

```
(pair? (cons x y)) = #t
(null? (cons x y)) = #f
        ⋮
```

For program correctness, any implementation of `cons`, `car`, and `cdr` that satisfies these laws is as good as any other. Exercise 16 on page 160 challenges you to come up with an unusual implementation.

3.4.2 Classifying operations

How can we tell when we have "enough" laws to describe a particular data type T? We examine the *operations* that work with type T.

- *Creators* and *producers* create new values of type T. Creators take arguments that are not of type T and produce a new value of type T. Not every creator requires arguments; for example, `'()` is a primitive, nullary creator for lists. Producers take arguments that are of type T, and possibly additional arguments. The primitive `cons` is a producer for lists.

  Creators and producers are sometimes grouped into a single category called *constructors*, but "constructor" is a slippery word. Our use of the word comes from algebraic specification, but the word "constructor" is also used in functional programming and in object-oriented programming—and each community means something different by it.

- An *observer* takes an argument of type T and produces a new value, which may or may not be of type T. An observer "looks inside" its argument and produces some fact about the argument, or perhaps some constituent value. Observers are sometimes also called *selectors* or *accessors*. Primitives `car`, `cdr`, `pair?`, and `null?` are observers for lists.

- By definition, creators, producers, and observers have no side effects. Operations that do have side effects are called *mutators*. They, too, can fit into the discipline of algebraic specification, but accounting for mutators makes specifications complicated. Writing algebraic specifications of mutators is beyond the scope of this book, but if you're interested, read the excellent book by Liskov and Guttag (1986).

We use this classification of operations to help decide how many laws are enough: we have enough laws for a type if the laws specify the results of all combinations of observers applied to creators and producers. We don't yet have enough laws for lists; the laws above don't specify what happens when we apply observers to the empty list. The laws for observing empty lists are as follows:

```
(pair? '()) = #f
(null? '()) = #t
  (car '()) is an error
  (cdr '()) is an error
```

Because S-expressions include not only lists but symbols, literals, etc., they have lots of creators, producers, and observers. All these functions lead to an overwhelming number of laws; we sketch only some Boolean laws below.

3.4.3 Boolean laws

For the Booleans, we consider the creators `#t` and `#f`, the producer `not`, and the observer `if`. The following laws on Booleans hold in any language, provided of course that all the expressions involved are free of side effects.

```
(if #t x y) = x
(if #f x y) = y
(if (not p) x y) = (if p y x)
```

These laws are all we need when `if` observes its first child. But if all three children of `if` are Booleans, we get more laws.

```
(if p #t #f) = p
(if p #f #t) = (not p)
```

3.4.4 More functions, more laws

Any time we define new functions, if those functions play the roles of creators, producers, or observers, we can get more laws. Adding functions `and` and `or` creates more laws on Booleans. Adding `append` creates new list laws. Such laws are not needed to *define* these new functions, because we already have definitions in terms of existing creators and producers. But some of the laws can be quite useful to simplify programs. In my own programming, for example, I often use these laws:

```
(and p q) = (and q p)
(and p #t) = p
(and p #f) = #f
(or p q) = (or q p)
(or p #t) = #t
(or p #f) = p
(append '() xs) = xs
(append (cons x '()) xs) = (cons x xs)
(append (append xs ys) zs) = (append xs (append ys zs))
```

3.4.5 Proving properties of programs

Laws about primitive operations must be taken as given; in fact, it's quite reasonable to take a set of laws as the *definition* of a set of primitives. But laws about functions like `append` don't have to be taken as given; by looking at the definition of `append`, we can prove such laws. A simple but powerful proof technique involves only expanding the definitions of functions and substituting equals for equals. This technique is called *equational reasoning*, and the resulting proofs are sometimes called *calculational proofs*.

As an example of the substitution of equals for equals, the "`null?`-empty law" tells us that `(null? '())` is equal to `#t`, and therefore one expression can be substituted for another. Another common substitution is to substitute actual parameters for formal parameters in the definition of a function.[6]

The following proof of the `append`-empty law first subtitutes for parameters, then applies a law about `null?` and a law about `if`:

```
(append '() xs)
  = {substitute actual parameters in definition of append}
(if (null? '())
    xs
    (cons (car '()) (append (cdr '()) xs)))
  = {null?-empty law}
(if #t
    xs
    (cons (car '()) (append (cdr '()) xs)))
  = {if-#t law}
xs
```

A single step in the proof is a single equality, presented in this form:

$$\begin{array}{l} term_1 \\ \quad = \{\textit{justification that } term_1 = term_2\} \\ term_2 \end{array}$$

These steps are chained together to show that every term is equal to every other term, and in particular that the first term is equal to the last term. In the example above, the chain of equalities establishes that

`(append '() xs)` = `xs`.

[6]The substitution of actual parameters for formal parameters may change the number of times each actual parameter appears, as well as the order in which the parameters may be evaluated. If the actual parameters can be evaluated without side effects, such a substitution is valid. If evaluating an actual parameter might result in a use of `set` or a call to `print`, substitution might change the outcome of the computation. For this reason, we typically use equational reasoning only on pure functional programs, which don't use `set` or `print`.

Here is a proof of another law: appending a singleton list is equivalent to `cons`:

```
(append (cons x '()) ys)
  = {substitute actual parameters in definition of append}
(if (null? (cons x '()))
    ys
    (cons (car (cons x '())) (append (cdr (cons 'x '())) ys)))
  = {null?-cons law}
(if #f
    ys
    (cons (car (cons x '())) (append (cdr (cons 'x '())) ys)))
  = {if-#f law}
(cons (car (cons x '())) (append (cdr (cons 'x '())) ys))
  = {car-cons law}
(cons x (append (cdr (cons 'x '())) ys))
  = {cdr-cons law}
(cons x (append '() ys))
  = {append-empty law}
(cons x ys)
```

Another simple law is that the length of a `cons` cell is one more than the length of the second argument, or

```
(length (cons x xs)) = (+ 1 (length xs))
```

Here's the proof:

```
(length (cons x xs))
  = {substitute actual parameter in definition of length}
(if (null? (cons x xs)) 0
    (+ 1 (length (cdr (cons x xs)))))
  = {null?-cons law}
(if #f 0
    (+ 1 (length (cdr (cons x xs)))))
  = {if-#f law}
(+ 1 (length (cdr (cons x xs))))
  = {cdr-cons law}
(+ 1 (length xs))
```

A more ambitious law, like the associativity of `append`, requires a more ambitious proof technique: induction. We can prove facts about lists and S-expressions by using *structural induction*:

- Prove the law holds for every base case. In the case of a list, we would prove that the law holds when the list is empty.
- Prove every induction step by assuming the law holds for the constituents. Again in the case of a list, we would prove the case for a nonempty list `xs` by assuming the induction hypothesis for the strictly smaller list `(cdr xs)`.

As an example, we prove the law

```
(length (append xs ys)) = (+ (length xs) (length ys))
```

The proof is by structural induction on `xs`. We begin with the base case in which `xs` is *nil*:

```
(length (append '() ys))
  = {append-empty law}
(length ys)
  = {zero is the additive identity}
(+ 0 (length ys))
  = {if-#t law, from right to left}
(+ (if #t 0 (+ 1 (length (cdr '())))) (length ys))
  = {null-empty law, from right to left}
(+ (if (null? '()) 0 (+ 1 (length (cdr '())))) (length ys))
  = {substitute actual parameter '() in definition of length}
(+ (length '()) (length ys))
```

In the induction step, we assume `xs` is not *nil*, and therefore that there exist a `z` and `zs` such that `xs` = `(cons z zs)`.

```
(length (append xs ys))
  = {by assumption that xs is not nil, xs = (cons z zs)}
(length (append (cons z zs) ys))
  = {substitute actual parameters in definition of append}
(length
  (if (null? (cons z zs))
      ys
      (cons (car (cons z zs)) (append (cdr (cons z zs)) ys))))
  = {null?-cons law}
(length
  (if #f
      ys
      (cons (car (cons z zs)) (append (cdr (cons z zs)) ys))))
  = {if-#f law}
(length
   (cons (car (cons z zs)) (append (cdr (cons z zs)) ys)))
  = {car-cons law}
(length (cons z (append (cdr (cons z zs)) ys)))
  = {cdr-cons law}
(length (cons z (append zs ys)))
  = {substitute actual parameter in definition of length}
(if (null? (cons z (append zs ys))) 0
    (+ 1 (length (cdr (cons z (append zs ys))))))
  = {null?-cons law}
(if #f 0
    (+ 1 (length (cdr (cons z (append zs ys))))))
  = {if-#f law}
(+ 1 (length (cdr (cons z (append zs ys)))))
  = {cdr-cons law}
(+ 1 (length (append zs ys)))
  = {the induction hypothesis}
(+ 1 (+ (length zs) (length ys)))
  = {associativity of +}
(+ (+ 1 (length zs)) (length ys))
  = {length-cons law}
(+ (length (cons z zs)) (length ys))
  = {by the initial assumption that xs = (cons z zs)}
(+ (length xs) (length ys))
```

These examples should teach you enough so that you can do Exercises 26 through 36 starting on page 165.

3.5 Language II: Local variables and `let`

Nobody writes big programs without local variables. In an imperative language like C or Impcore, you can introduce a local variable without initializing it, and it's common to assign to a single local variable more than once. In Scheme, local variables are *always* initialized when introduced, and afterward, they are rarely changed. The best practice is to introduce a local variable in order to *name the results of a computation*, then never assign to that variable again. Throughout its lifetime, such a variable stands for a single value. When a variable always stands for the same value, code that uses the variable is easier to understand.

In Scheme, we introduce local variables by *let binding*. The idea is to "**let** $x = e'$ **in** e," which roughly means "evaluate e', let x stand for the resulting value, and evaluate e." Proper Scheme syntax binds a collection of values:

```
(let ((x1 e1) (x2 e2) ... (xn en)) e).
```

The `let` expression is the first expression of μScheme that isn't also in Impcore. It is evaluated as follows: First evaluate $\texttt{e}_1$ through $\texttt{e}_n$. Suppose the results are $v_1, \ldots, v_n$. Now, *extend* the local environment with bindings $\texttt{x}_1 \mapsto v_1, \ldots, \texttt{x}_n \mapsto v_n$, and finally evaluate `e` in the extended environment. The result of the entire `let` expression is the result of evaluating `e`.

In its simplest form, `let` both improves readability and avoids repeating computation. As an example, suppose we want to compute the solutions of the equation $ax^2 + bx + c = 0$. From high-school algebra, we can use the quadratic formula: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Although this formula is not much use without real numbers, we can still implement it in μScheme and use the implementation to solve such equations as $x^2 + 3x - 70 = 0$.

82a ⟨*transcript* 66⟩+≡ ◁75b 82b▷

```
⟨definition of sqrt 603a⟩
-> (define roots (a b c)
     (let ((discriminant (sqrt (- (* b b) (* 4 (* a c)))))
           (two-a        (* 2 a))
           (minus-b      (- 0 b)))
       (list2 (/ (+ minus-b discriminant) two-a)
              (/ (- minus-b discriminant) two-a))))
-> (roots 1 3 -70)
(7 -10)
```

In a `let` expression, all of the $\texttt{e}_i$'s are evaluated before any of the $\texttt{x}_i$'s are bound. It is often more useful to evaluate and bind in sequence, so that the expression $\texttt{e}_i$ can refer to all the named values $\texttt{x}_1, \ldots, \texttt{x}_{i-1}$. We can do so with a `let*` expression, which is exactly like a `let` expression, except it first evaluates $\texttt{e}_1$, then extends the environment by binding the result to $\texttt{x}_1$. Then it evaluates $\texttt{e}_2$ in the extended environment, and so on.

list2 125b
sqrt 603a

To understand the difference between `let` and `let*`, look at these examples:

82b ⟨*transcript* 66⟩+≡ ◁82a 83a▷

```
-> (val x 'global-x)
-> (val y 'global-y)
-> (let
     ((x 'local-x)
      (y x))
     (list2 x y))
(local-x global-x)
```

Because the expressions in a `let` are evaluated in the original environment, variables introduced in one part of a `let` cannot influence the definition of variables introduced in another part. In this example, the `x` in `(y x)` refers to the *global* definition of `x`.

Here is the same code using `let*`:

83a ⟨*transcript* 66⟩+≡ ◁82b 83b▷

```
-> (val x 'global-x)
-> (val y 'global-y)
-> (let*
     ((x 'local-x)
      (y x))
     (list2 x y))
(local-x local-x)
```

Because the expressions in `let*` are evaluated and bound in sequence, in this example the `x` in `(y x)` refers to the *local* definition of `x`. If we did not have `let*`, we could simulate it with a nested sequence of `let` expressions, but `let*` is more convenient.

Scheme has yet another variety of let: `letrec`. In a `let` expression, none of the $\texttt{x}_i$'s can be used in any of the $\texttt{e}_i$'s. In a `let*` expression, each $\texttt{x}_i$ can be used in any $\texttt{e}_j$ with $j > i$, that is, an `x` can be used in all the `e`'s that follow it. In a `letrec` expression, *all* of the $\texttt{x}_i$'s can be used in *all* of the $\texttt{e}_i$'s, regardless of order. As you might imagine from the name, `letrec` is used primarily for defining recursive functions. The details are subtle, and we postpone them to Section 3.12.1. But as an example, here is a particularly inefficient pair of mutually recursive functions that determine whether a nonnegative number is even.

83b ⟨*transcript* 66⟩+≡ ◁83a 85a▷

```
-> (val even?
     (letrec
       ((odd-test (lambda (n) (not (even-test n))))
        (even-test
          (lambda (n)
            (if (= n 0)
              #t
              (odd-test (- n 1))))))
       even-test))
-> (even? 0)
#t
-> (even? 1)
#f
-> (even? 2)
#t
```

Exercise 40 on page 167 asks you to investigate the possibility of μScheme without `letrec`.

list2 125b
not 123a

3.6 Practice II: Recursive functions on binary trees

Scheme's S-expressions make it easy to manipulate trees; a tree can be represented as a list whose elements include other lists. But for a beginning Scheme programmer, it can be difficult to keep track of how data structures are represented, because everything is made with cons cells. You can tame your Scheme data structures, especially recursive data structures, by writing down equations that capture precisely what is meant by such terms as "tree," "list," or "S-expression."

Let's begin with lists. To describe the set of all lists whose elements are in set A, we'll write $LIST(A)$. Lists are made using *nil* and `cons` according to the following equation:

$$LIST(A) = \{\texttt{'()}\} \cup \{\texttt{(cons } a \ as\texttt{)} \mid a \in A \wedge as \in LIST(A)\} \tag{3.1}$$

Equation 3.1 is not a *definition* of $LIST(A)$; it is a *recursion equation.* The set $LIST(A)$ is the *smallest* set satisfying the equation.[7] Functions that consume lists must handle the cases for both `'()` and `cons`—as must proofs of facts about lists.

We can express the same information in another way, by writing a proof system for membership in the set $LIST(A)$. The judgment form is $v \in LIST(A)$:

$$\frac{}{\texttt{'()} \in LIST(A)} \tag{EMPTYLIST}$$

$$\frac{a \in A \quad as \in LIST(A)}{\texttt{(cons } a \ as\texttt{)} \in LIST(A)} \tag{CONSLIST}$$

A value v is in $LIST(A)$ if and only if there is a proof of $v \in LIST(A)$.

Scheme comes with some primitive sets of values, which we abbreviate $BOOL$, NUM, and SYM. These sets, together with *nil*, form the *atoms*:

$$ATOM = BOOL \cup NUM \cup SYM \cup \{\texttt{'()}\} \tag{3.2}$$

A general S-expression is an atom or a pair of S-expressions:

$$SEXP = ATOM \cup \{\texttt{(cons } v_1 \ v_2\texttt{)} \mid v_1 \in SEXP \wedge v_2 \in SEXP\} \tag{3.3}$$

(This equation leaves procedures out of the picture.)

We can now say precisely what we mean by a "tree." More precisely, we define a set of *labeled binary trees* with labels drawn from set L. We represent a node with no children by a bare label. We represent a node with children as a triple (a 3-element list) containing a label and two subtrees.

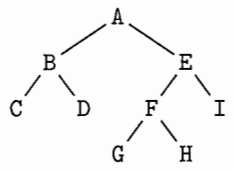
$$BINTREE(L) = L \cup \{\texttt{(list3 } l \ t_1 \ t_2\texttt{)} \mid l \in L \wedge t_1 \in BINTREE(L) \wedge t_2 \in BINTREE(L)\} \tag{3.4}$$

The representation defined by this equation is very strange—we can't represent an empty tree or a node with only one child—but it makes it very easy to read and write S-expression literals that represent trees. For example, the list

84 ⟨*none* 84⟩≡

```
'(A (B C D) (E (F G H) I))
```

represents the tree



[7] This set is the intersection of all sets satisfying the equation.

The mathematical definition of $BINTREE(L)$ also presents a practical problem: if the label set L includes any three-element lists, it will be hard for our code to tell the difference between a childless node and a node with two children. For that reason, we will restrict ourselves to trees labeled with atoms, which are in the set $BINTREE(ATOM)$. Given this restricted set, an atom always represents a childless node, and a non-atom always represents a node with two children.

Are we ready to write code? Not yet. We don't want to confuse trees with *other* data structures that are also created with atoms and `cons` and are observed using `car` and `cdr`. To eliminate the possibility of confusion, we don't use `atom?` or `car` or `cdr` directly. Instead, we bury these general-purpose functions inside other, tree-specific functions. The tree-specific functions are inspired by Equation 3.4. If only the tree-specific functions are allowed to manipulate trees, we can change our representation later without having to change anything but these functions. The tree-specific functions also make it easier to read our tree-manipulation code.

85a ⟨*transcript* 66⟩+≡ ◁83b 85b▷

```
-> (define leaf? (node) (atom? node))
-> (define left  (node) (cadr  node))
-> (define right (node) (caddr node))
-> (define label (node) (if (leaf? node) node (car node)))
```

Now we're ready to write code. We present functions for pre-order traversal and level-order traversal. The function for level-order traversal illustrates `let*`. Just as in the case of lists, these functions have a recursive structure that is dictated by the structure of the set $BINTREE(ATOM)$:

- There are two kinds of tree: a leaf, and a tree with two children. A tree-consuming function begins with case analysis; it uses `leaf?` to tell what kind of tree it has.
- If the tree `t` is a leaf, the function includes code for the base case of its algorithm.
- If the tree `t` is not a leaf, the function includes code for its induction step. Because a non-leaf has both a left and a right child, the induction step most likely includes recursive calls on both `(left t)` and `(right t)`.

As an example, here is a pre-order traversal; the function `pre-ord` takes a tree `t` and returns a list containing the nodes of `t` in preorder.

85b ⟨*transcript* 66⟩+≡ ◁85a 86a▷

```
-> (define pre-ord (t)
     (cons
       (label t)
       (if (leaf? t)
         '()
         (append
           (pre-ord (left t))
           (pre-ord (right t))))))
-> (pre-ord '(A (B C D) (E (F G H) I)))
(A B C D E F G H I)
```

append 70a
atom? 69a
caddr 125a
cadr 125a
car $\mathcal{P}$
cons $\mathcal{P}$

Function `pre-ord` confirms an oddity of our representation: there is no way to represent an empty tree, and just as we would expect, there is no way that `pre-ord` can return an empty list.

Level-order, or breadth-first, traversal—where the tree is visited one level at a time, producing the result `(A B E C D F I G H)` for our example—is trickier. The basic method is to keep a queue of nodes that are yet to be visited. When the queue is empty, the traversal is finished. Otherwise, visit the node at the front of the queue, then place its children at the end of the queue.

We begin with operations that manipulate queues.

86a ⟨*transcript* 66⟩+≡ ◁85b 86b▷

```
-> (val emptyqueue '())
-> (define front         (q) (car q))
-> (define without-front (q) (cdr q))
-> (define enqueue (t q)
     (if (null? q)
       (list1 t)
       (cons (car q) (enqueue t (cdr q)))))
-> (define empty? (q) (null? q))
```

This implementation of queues is woefully inefficient, but it's simple. Exercise 17 describes a more efficient implementation.

The function `level-ord` performs the traversal by calling the auxiliary function `level-ord*`, passing it a queue whose one entry is the root node of the tree.

86b ⟨*transcript* 66⟩+≡ ◁86a 87a▷

```
-> (define level-ord* (queue)
     (if (empty? queue)
       '()
       (let* ((hd   (front         queue))
              (tl   (without-front queue))
              (newq (if (leaf? hd)
                        tl
                        (enqueue (right hd) (enqueue (left hd) tl)))))
         (cons (label hd) (level-ord* newq)))))
-> (define level-ord (t)
     (level-ord* (enqueue t emptyqueue)))
-> (level-ord '(A (B C D) (E (F G H) I)))
(A B E C D F I G H)
```

We exploit the sequential semantics of `let*` to define `hd` in an early binding, then refer to `hd` in a later binding (of `newq`).

car	$\mathcal{P}$
cdr	$\mathcal{P}$
cons	$\mathcal{P}$
label	85a
leaf?	85a
left	85a
list1	125b
null?	$\mathcal{P}$
right	85a

The code above is correct, but it is also error-prone. For example, it is easy to use `queue` in place of `tl`, causing an infinite loop. The imperative way to avoid this error is to have just one variable, `queue`, and to keep assigning to it. The applicative way is to repeatedly *rebind* the name, so that new bindings hide previous versions.

87a ⟨*transcript* 66⟩+≡ ◁86b 87b▷

```
-> (define level-ord* (queue)
     (if (empty? queue)
       '()
       (let* ((hd    (front         queue))
              (queue (without-front queue))
              (queue (if (leaf? hd)
                         queue
                         (enqueue (right hd) (enqueue (left hd) queue)))))
         (cons (label hd) (level-ord* queue)))))
-> (define level-ord (t)
     (level-ord* (enqueue t emptyqueue)))
-> (level-ord '(A (B C D) (E (F G H) I)))
(A B E C D F I G H)
```

Whether you use imperative style or applicative style is often a matter of taste, but either is better than letting names proliferate.

3.7 Language III: First-class functions, `lambda`, and locations

In Impcore, functions are special things that are different from values. As in C, they can be defined only at top level. By providing one new language construct, Scheme removes these restrictions. In Scheme, the *lambda* expression

```
(lambda (x y ... z) e)
```

denotes the function that takes values $x, y, \ldots, z$ and returns the result of evaluating `e` with `x` bound to x, `y` bound to y, and so on. The `lambda` expression is the second kind of expression that appears in μScheme but not in Impcore. The `lambda` expression is extraordinarily powerful; it is the foundation on which many useful programming techniques are built. The most important and widely used of those techniques are presented in this chapter.

The power of `lambda` lies in the many ways it is deployed; by itself, `lambda` should seem simple and innocuous. For example, `(lambda (x y) (+ (* x x) (* y y)))` denotes the function that, given x and y, returns $x^2 + y^2$. This function is *anonymous*; unlike Impcore functions and C functions, Scheme functions need not be named.

`cons`	$\mathcal{P}$
`empty?`	86a
`emptyqueue`	86a
`enqueue`	86a
`front`	86a
`label`	85a
`leaf?`	85a
`left`	85a
`right`	85a
`without-front`	86a

87b ⟨*transcript* 66⟩+≡ ◁87a 89a▷

```
-> ((lambda (x y) (+ (* x x) (* y y))) 707 707)
999698
-> ((lambda (x y z) (+ x (+ y z))) 1 2 3)
6
-> ((lambda (y) (* y y)) 7)
49
```

In the sections above, we defined functions without saying "`lambda`," so you might think that μScheme has two kinds of functions, one created with `lambda` and one created with `define`. But `define` is simply a convenient abbreviation, also called "syntactic sugar." We give the meaning of `define` by rewriting it to a definition in a core language having only `val` and `lambda`. The μScheme interpreter rewrites

```
(define f (x y ... z) e)
```

into

```
(val f (lambda (x y ... z) e)).
```

People who define programming languages love this kind of trick, because the trick makes it possible to provide a big "surface syntax" for programmers to use, while keeping the core language and its operational semantics very small. Here, the trick is possible only because in μScheme, functions and values live in the same environment (see Section 3.12 on page 115). In Impcore, where functions and values live in different environments, such a trick would be impossible.

Using `lambda` on the right hand side of a `val` definition is just one way to exploit the idea that a function defined with `lambda` is just another sort of value. What else do we do with values? Here's a classic list, which you can use to judge *any* programming language, of things that programmers do with values:

- Pass values as arguments to functions.
- Return values as results from functions.
- Store values in global variables, using `set`.
- Save values in data structures on the heap, which in Scheme we would do with `cons`.

If you can do all these kinds of things with a value, some people say that the value is *first-class*. Some people will even characterize Scheme and other functional languages by saying that they support "first-class functions." But the real key, as we'll see below, is that Scheme supports first-class, *nested* functions. Let's look at some examples.

If functions are first-class, then we can pass a μScheme function (defined with `define` or with `lambda`) as an argument to another function. What can the function that *receives* the argument do? It can do all the standard things that it can do with any first-class value, which are listed in the bullets above. But there is one more thing it can do—the only *interesting* thing to do with a function—*apply* it. Here' an example of a function that receives another function `f` as a parameter, then applies `f`. In addition to a function `f`, the function `apply-n-times` receives an integer `n`, and an argument `x`; it then applies `f` to `x` `n` times.

89a ⟨*transcript* 66⟩+≡ ◁87b 89b▷

```
-> (define apply-n-times (n f x)
     (if (= 0 n)
        x
        (apply-n-times (- n 1) f (f x))))
-> (define twice  (n) (* 2 n))
-> (define square (n) (* n n))
-> (apply-n-times 2 twice 10)
40
-> (apply-n-times 2 square 10)
10000
-> (apply-n-times 10 twice 1)
1024
-> (apply-n-times 10 square 1)
1
```

There's a handy piece of jargon that goes with a function like `apply-n-times`: a function that takes another function as an argument is called a *higher-order function.* The term "higher-order function" is inclusive; it also describes a function that *returns* another function. Functions like `twice` and `square`, which neither take functions as arguments nor return functions as results, are called *first-order functions.*

The inclusivity of the term "higher-order function" is actually a pity, because in programming languages, the important distinction is not whether a function can take another function as an argument, but exactly when a function can return another function as a result. For example, the function `apply-n-times`, which takes one function argument but returns no function results, is a higher-order function of the less interesting kind. To take a function as an argument, we don't really need `lambda`, and if you look at the code, you'll see there's no `lambda` in the body; the only `lambda` in play is the one used to implement `define`. Because it doesn't use the full power of `lambda`, `apply-n-times` could be implemented in many other languages, including C. (C functions are first-class and can therefore be passed as arguments to other functions.[8]) A really interesting `lambda` expression is evaluated *inside* the body of another function, and it produces a closure which is returned, or which goes into a data structure that is returned. This kind of interesting `lambda` creates a new function anonymously, "on the fly." Here is a contrived example:

89b ⟨*transcript* 66⟩+≡ ◁89a 90▷

```
-> (val add (lambda (x) (lambda (y) (+ x y))))
-> (val add1 (add 1))
-> (add1 4)
5
```

[8] Actually, in C it is *pointers* to functions that are the first-class values, but this distinction makes no practical difference.

Here `add` is defined by the outer `lambda` expression, which takes `x` as an argument and is not so interesting. But the inner `lambda`, which takes `y` as an argument, is one of the interesting ones: every time it is evaluated, it creates a new function.

Here's how the two `lambda`s work together: given argument m, `add` returns a function which, given argument n, returns $m+n$. Thus, `add1` is a function which, given argument n, returns $1+n$. By applying `add` to different integers, we can create *arbitrarily many* functions. Not only `(add 1)` but `(add 2)` and `(add 100)` can be functions. Even an expression like `(add (length xs))` returns a function.

In the `add` example, the key property is that the `(lambda (y) (+ x y))` is nested inside the outer `lambda` that defines `add`. Every time `add` is called, the inner `lambda` is evaluated, and in Scheme, every time a `lambda` expression is evaluated, a new function is created. In languages like Impcore and C, which don't have `lambda`, a new function can be created only by writing it explicitly in the source code.

3.7.1 Closures

In the `add` example, the expression `(lambda (y) (+ x y))` can stand for more than one function, depending on what `x` stands for. To make a value that represents one of these functions, we pair the `lambda` expression with an environment that says what `x` stands for. Such a pair is called a *closure*. We write a closure in "banana brackets," as in $(\!|\texttt{(lambda (y) (+ x y))}, \{\texttt{x} \mapsto 1\}|\!)$. The banana brackets emphasize that a closure cannot be written directly using Scheme syntax; the closure is a value that Scheme builds from a `lambda` expression. Closures are widely used in interpreters and compilers, and even in some kinds of semantics.

The pairing idea—a `lambda` expression together with an environment—is the key idea behind closures, and it provides the simplest explanation of how to implement first-class, nested functions. But even once we're committed to closures, as language designers we still have a decision left to make, and that decision revolves around the environment part of the closure. The decision is simple: what sort of thing should a name like `x` stand for? In particular, if we evaluate a `lambda` to produce a closure, and afterward some other code changes `x`, should the closure be affected somehow? Here's an example in which the answer is yes:

90 ⟨*transcript* 66⟩+≡ ◁89b 91a▷

```
-> (val counter-from
     (lambda (x)
       (lambda () (set x (+ x 1)))))
-> (val ten (counter-from 10))
<procedure>
-> (ten)
11
-> (ten)
12
-> (ten)
13
```

What is the value of `ten`? It is the result of evaluating `(lambda () (set x (+ x 1)))` in an environment where `x` is 10. In the simple model above, `ten` must therefore stand for the closure $\langle\!|$`(lambda () (set x (+ x 1)))`$, \{\texttt{x} \mapsto 10\} |\!\rangle$. But then how do we account for `x` changing as we keep calling `ten`? We would somehow have to change the environment in the closure to $\{\texttt{x} \mapsto 11\}$, then $\{\texttt{x} \mapsto 12\}$, and so on. The semantics of `set` would have to be pretty complicated, and it wouldn't be so easy to implement, either. For example, if we wanted to provide another function to reset `x` to zero, the `set` executing inside one closure would have to change the environment of an entirely different closure. One way around this problem is to avoid it entirely; for example, we could eliminate `set`. But if we want to keep `set`, there is a simple way out of the dilemma, provided we are willing to introduce a new semantic idea: mutable *locations*.

Mutable locations in closures

A location is a container that holds a value. In Scheme, a named variable always stands for a location, never for a value. The truth about environments is that an environment binds names to locations, not to values. Using a variable means looking up its name in the environment, then seeing what value is in the location. Assigning to a variable means changing the contents of the location, *not* changing anything about the environment. This is a simple abstraction, and it is a good match for real hardware, where we can represent a location as a pointer, a memory address, or even a hardware register. When we evaluate a `lambda` expression, the environment that goes into the closure points to locations, not directly to values. For example, the value of `ten`, is the closure $\langle\!|$`(lambda () (set x (+ x 1)))`$, \{\texttt{x} \mapsto \ell\} |\!\rangle$, where ℓ is a location. The location ℓ *initially* contains 10, but every time the internal expression `(set x (+ x 1))` is evaluated, the number contained in location ℓ is increased by 1.

Now we can craft an example in which two different closures refer to the same mutable location. This situation, in which two or more `lambda` expressions refer to the same variable, *and* the variable is mutated, is an example of *shared mutable state*. To craft the example, we build a counter which provides not only a function that increments the counter, but also a function that resets the counter to zero.

91a ⟨*transcript* 66⟩+≡ ◁90 91b▷

```
-> (val resettable-counter-from
     (lambda (x)
       (list2
         (lambda () (set x (+ x 1)))
         (lambda () (set x 0)))))
```

The two innermost `lambda` expressions must refer to the *same* `x`, so that when we reset the counter, the stepper function starts over at 0. But two *different* applications of `resettable-counter-from` must refer to *different* xs, so that two independent counters never interfere.

cadr 125a
car P
list2 125b

Function `resettable-counter-from` satisfies both properties. It evaluates the two `lambda`s, saves them in a data structure on the heap (created with `list2`), and returns a list of two functions: the step function and the reset function. To make it convenient to use the resettable counter, we define auxiliary functions that implement the step and reset operations.

91b ⟨*transcript* 66⟩+≡ ◁91a 92a▷

```
-> (val step  (lambda (counter) ((car  counter))))
-> (val reset (lambda (counter) ((cadr counter))))
```

Because the functions stored in the `counter` take no parameters, it is easy to overlook the double applications in the definitions of `step` and `reset`. An inner application, e.g., `(car counter)`, applies a primitive to the list `counter`, extracting a function. An outer application, e.g., `((car counter))`, applies that extracted, parameterless function.

A transcript shows that we can create two counters `hundred` and `twenty` which count independently and can both be reset.

92a ⟨*transcript* 66⟩+≡ ◁91b 92b▷

```
-> (val hundred (resettable-counter-from 100))
-> (val twenty  (resettable-counter-from 20))
-> (step hundred)
101
-> (step hundred)
102
-> (step twenty)
21
-> (reset hundred)
0
-> (step hundred)
1
-> (step twenty)
22
```

Using closures for private, persistent mutable state

Scheme closures with mutable locations are good for more than just sharing mutable state among functions. Closures can also be used to keep mutable state that is *private* to a function but also *persistent* across calls to that function. Languages that don't provide the full power of first-class, nested functions sometimes provide this ability as a special feature, usually called "own variables." An own variable is local to a function, but its value is preserved across calls of the function. The name comes from Algol-60, but C also provides own variables, which it defines using the keyword `static`. As an example of a function that uses private, persistent mutable state, we'll define a very simple pseudo-random number generator.

An *applicative pseudorandom number generator*, which we'll call `arand`, is a function that is applied to a number and produces another number. The input number is called the *seed*. The generator is "pseudorandom" if without seeing the code, there's no easy way to predict the output number from the input number.[9] It's a rare algorithm that consumes only one random number, and in practice, the output of a random-number generator is used not only as a source of randomness but also as a new seed. In effect, an applicative random-number generator is applied repeatedly to its own outputs, as in `apply-n-times`. The result is a *sequence* of pseudorandom numbers. Also in practice, a random-number generator is considered good only if a long sequence of its outputs cannot be distinguished from truly random outputs, even by a bevy of statistical tests.

mod 123d
reset 91b
resettable-counter-from 91a
step 91b

Here's an applicative pseudorandom-number generator which operates on integers in the range 0 to 1023. It uses the *linear congruential* method (Knuth 1981, pp. 9–25). This generator is not, in any statistical sense, good. For one thing, after generating only 1024 different numbers, it starts repeating.

92b ⟨*transcript* 66⟩+≡ ◁92a 93▷

```
-> (val arand (lambda (seed) (mod (+ (* seed 9) 5) 1024)))
```

[9] Because it's a mouthful, we usually drop the "pseudo."

The interface to `arand` requires that if an application wants a sequence of random numbers, it must keep track of the number most recently returned by `arand`, so it can use that number as a seed for the next call to `arand`. Because the current value of the seed must be passed around among all functions that consume random numbers, any module that uses `arand` must know about every other module that uses `arand`, and the interfaces of all these modules must provide for the seed to be passed. Such an organization violates everything programmers know about encapsulation (hiding the fact that a module consumes random numbers) and modularity (preventing decisions made in one module from affecting code in another module). The standard solution to this problem is to create an *imperative* random-number generator, which we'll call `irand`, which keeps the current seed in a mutable location and which is called without any parameters.

In some languages, the only way to make an imperative random-number generator is to make the seed a global variable. But the seed is private information, and if it is corrupted, it can break a program in the worst possible way: the program may produce *plausible* wrong answers. And if there is a single global seed, it becomes impossible to debug one module by *replaying* two versions of the code with the same sequence of pseudorandom numbers: if another module asks for a random number, the sequence that the first module sees will be different.

We can solve both the protection problem and the independent-debugging problem by putting the seed into a closure. If the seed is available *only* through the closure, then only `irand` has access, and no other module can corrupt it. And if we are making closures, it is easy to make multiple random-number generators, which can be used independently for debugging.

Here's the code.

93 ⟨*transcript* 66⟩+≡ ◁92b 94b▷

```
-> (val mk-irand (lambda (arng seed)
                   (lambda () (set seed (arng seed)))))
-> (val irand (mk-irand arand 1))
-> (irand)
14
-> (irand)
131
-> (irand)
160
-> (val repeatable-irand (mk-irand arand 1))
-> (repeatable-irand)
14
-> (irand)
421
```

The value of `seed` is updated at each call to `irand`, but it is protected from access by anyone except `irand`. The function `repeatable-irand`, which might be used to replay an execution for debugging, repeats the *same* sequence $[1, 14, 131, 160, 421, \ldots]$ no matter what happens with `irand`.

`arand` 92b

We get one more benefit from using higher-order functions: if we discover a better algorithm for generating random numbers, we don't rewrite any code—we simply define a new *applicative* pseudorandom-number generator, and we pass it, along with a seed, to `mk-irand`.

3.7.2 Simple, useful higher-order functions

At this stage we've seen some simple examples of `lambda`, we've looked at how `lambda` is implemented using closures, and we've exploited the semantics of closures to implement both shared mutable state and private mutable state. But `lambda` offers more than just tricks with mutable state. Perhaps the biggest reason that functional programmers are excited about `lambda` is that `lambda` enables us to program not just algorithms but *patterns of computation.* The idea of a "pattern of computation" probably seems very vague and abstract to you now, but in the next few sections, we'll go deep into this idea and see lots of examples.

We've seen one very minor example already: the function `mk-irand` could be viewed as a pattern of computation, which says "if you know a good pure function for producing pseudorandom numbers, make an imperative random-number generator out of it." This pattern of computation, while handy, is not something we expect to use a lot. In the next few sections, we explore patterns of computations that we *do* expect to use a lot; in fact, based on years of experience and observation, we *know* that they can be used a lot. We start very simply with code that just uses `lambda` to make new functions out of old functions. Then in the next section we tackle higher-order functions that embody common ways of programming functions that consume lists.

Composition

One of the simplest higher-order functions is `o` (pronounced "circle" or even "compose"), which returns the composition of two one-argument functions, often written $f \circ g$:

94a ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁74b 95d▷

```
(define o (f g) (lambda (x) (f (g x))))
```

In large programs, function composition is often used to improve modularity: an algorithm can be broken down into pieces that are connected using function composition. There are some extended examples in Chapter 8. Even as you're starting out, however, you can find some simple ways to use function composition. One of the most common uses is to negate a predicate by composing `not` with it:

94b ⟨*transcript* 66⟩+≡ ◁93 95a▷

```
-> (define even? (n) (= 0 (mod n 2)))
-> (val odd? (o not even?))
-> (odd? 3)
#t
-> (odd? 4)
#f
```

mod 123d
not 123a

Currying and partial application

The functions `+` and `add` that we have defined differ in that `+` takes both of its arguments at once, while `add` takes one argument at a time: `(add 1)` is a function that adds `1` to its argument, while `(+ 1)` is an error. Similarly, `(+ 1 2)` is `3`, while `(add 1 2)` is an error. But `add` and `+` are really two forms of the same function; they differ only in the way they take their arguments. We call `add` the *curried* form of the function, and we call `+` the *uncurried* form. The name honors the logician Haskell B. Curry.

Any function can be put into curried form; a curried function simply takes its arguments one at a time. If it needs more than one argument, it takes the first argument, then returns a new function that expects the remaining arguments, also one at a time. For example, here is a curried form of the `list3` function:

95a ⟨*transcript* 66⟩+≡ ◁94b 95b▷

```
-> (val list3-curried (lambda (a) (lambda (b) (lambda (c) (list3 a b c)))))
-> (list3-curried 'x)
<procedure>
-> ((list3-curried 'x) 'y)
<procedure>
-> (((list3-curried 'x) 'y) 'z)
(x y z)
```

As you can see, the syntax of μScheme is unkind to currying; you need lots of `lambda`s to define a curried function and lots of parentheses to apply it.[10] If you're using a μScheme interpreter interactively, you can reduce the notational burden by using the convention that value of the last expression typed in at top level is bound to variable `it`:

95b ⟨*transcript* 66⟩+≡ ◁95a 95c▷

```
-> (list3-curried 'a)
<procedure>
-> (it 'b)
<procedure>
-> (it 'c)
(a b c)
```

If currying is so awkward, why bother with it? Because the *partial application* of a curried function may be useful. A curried function is partially applied if you apply it to only *some* of its arguments, then save the resulting function to apply later. Here we have a curried function of `<`, which we partially apply to 0. At that point we have a function that takes any m and tells us if $0 < m$:

95c ⟨*transcript* 66⟩+≡ ◁95b 96a▷

```
-> (val <-curried (lambda (n) (lambda (m) (< n m))))
-> (val positive? (<-curried 0))
-> (positive? 0)
#f
-> (positive? 8)
#t
-> (positive? -3)
#f
```

Because these curried binary functions are so useful, μScheme provides a way to convert binary functions between their curried and uncurried forms.

95d ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁94a 99a▷ `list3` 125b

```
(define curry   (f) (lambda (x) (lambda (y) (f x y))))
(define uncurry (f) (lambda (x y) ((f x) y)))
```

[10] More recently developed functional languages, like Standard ML, OCaml, and Haskell, are very friendly to currying, so much so that in Caml and Haskell it is standard for *all* functions to be defined in curried form.

96a ⟨*transcript* 66⟩+≡ ◁95c 96b▷

```
-> (val zero? ((curry =) 0))
-> (zero? 0)
#t
-> (val add1 ((curry +) 1))
-> (add1 4)
5
-> (val new+ (uncurry (curry +)))
-> (new+ 1 4)
5
```

If currying makes your head explode, don't panic—I expect it. Both currying and composition are going to be easier to understand when we start to deploy them with more interesting data structures, like lists.

3.8 Practice III: Higher-order functions on lists

If higher-order functions embody "patterns of computation," what are those patterns? The first really convincing examples are patterns of computation on lists. Some of the most popular and widely reused patterns can be described informally as follows:

- Search a list for an element.
- Pick just some elements from a list.
- From an old list, make a new list by transforming each element.
- Check every element of a list to make sure it is OK.
- Visit every list element, perform a computation there, and accumulate a result. Or more loosely, do something to every element of a list.

These patterns, and more besides, are embodied in the higher-order functions `exists?`, `filter`, `map`, `all?`, and `foldr`, all of which are in the initial basis of μScheme. (If you've heard of Google MapReduce, it's the same `map`, and Reduce is a parallel variant of `foldr`.)

3.8.1 Standard higher-order functions on lists

add1 89b
curry 95d
filter 99a
mod 123d
seq 71c
uncurry 95d

Function `filter` takes a predicate `p?` and a list `xs`, and it returns a new list consisting of only those elements of `xs` that satisfy `p?`:

⟨*transcript* 66⟩+≡ ◁96a 97a▷

```
-> (define even? (x) (= (mod x 2) 0))
-> (filter even? (seq 1 10))
(2 4 6 8 10)
```

Exercise 8 on page 157 asks you to use `filter` to define a very concise version of the `remove-multiples` function from Section 3.3.3.

Two more standard higher-order functions, each of which also takes a predicate and a list as arguments, are `exists?` and `all?`. As you might expect, `exists?` tells whether there is an element of the list satisfying the predicate; `all?` tells whether they all do.

97a ⟨*transcript 66*⟩+≡ ◁96b 97b▷

```
-> (exists? even? (seq 1 10))
#t
-> (all? even? (seq 1 10))
#f
-> (all? even? (filter even? (seq 1 10)))
#t
-> (exists? even? (filter (o not even?) (seq 1 10)))
#f
```

When called on the empty list, an important "corner case," `exists?` and `all?` act like the mathematical ∃ and ∀.

97b ⟨*transcript 66*⟩+≡ ◁97a 97c▷

```
-> (exists? even? '())
#f
-> (all? even? '())
#t
```

Functions `filter`, `exists?`, and `all?` take only predicates as arguments, but `map` is a bit more interesting; `(map f xs)` returns the list formed by applying function `f` to every element of list `xs`.

97c ⟨*transcript 66*⟩+≡ ◁97b 98▷

```
-> (map add1 '(3 4 5))
(4 5 6)
-> (map ((curry +) 5) '(3 4 5))
(8 9 10)
-> (map (lambda (x) (* x x)) (seq 1 10))
(1 4 9 16 25 36 49 64 81 100)
-> (primes<= 20)
(2 3 5 7 11 13 17 19)
-> (map ((curry <) 10) (primes<= 20))
(#f #f #f #f #t #t #t #t)
```

Higher-order functions on lists can make programs wonderfully concise, because they relieve you from writing the same recursive list traversals over and over again. These recursive traversals are some of the patterns of computations I keep talking about. Having a concise way to write common traversals helps not just the person writing the code but also the people who read it. If you're reading code and you see a recursive function, you have to be prepared for anything. But if you see `map` or `filter`, you know exactly what computation is happening.

A very general traversal strategy is to combine the `car` of a list with the results of a recursive call. This strategy is embodied in the higher-order function `foldr`. To understand `foldr`, it helps to change notation, so let us temporarily write lists with an explicit infix `cons`, notated with a double colon (::). If a list `vs` has the form $v_1 :: v_2 :: \cdots :: v_n :: \mathit{nil}$, then `(foldr` $\oplus$ $\aleph$ `vs)` is the value $v_1 \oplus v_2 \oplus \cdots \oplus v_n \oplus \aleph$. In other words, imagine that we replace every `cons` with $\oplus$, and we replace *nil* with $\aleph$.

add1 89b
all? 99c
curry 95d
even? 96b
exists? 99c
filter 99a
map 99b
not 123a
primes<= 72a
seq 71c

But what if $\oplus$ is not an associative operator? The `r` in `foldr` means that we associate computation to the right. The right operand of $\oplus$ is always either $\aleph$ or the result of applying a previous $\oplus$:

$$\texttt{(foldr}\ \oplus\ \aleph\ \texttt{'(}v_1\ v_2\ \cdots\ v_n\texttt{))} = v_1 \oplus (v_2 \oplus (\cdots \oplus (v_n \oplus \aleph))).$$

Function `foldr` has a companion function, `foldl`, in which $\oplus$ associates to the left and $\aleph$ goes with v_1, not with v_n. Because each result becomes a right operand of $\oplus$, `foldl` is effectively `foldr` operating on a reversed list:

$$\texttt{(foldl}\ \oplus\ \aleph\ \texttt{'(}v_1\ v_2\ \cdots\ v_n\texttt{))} = v_n \oplus (v_{n-1} \oplus (\cdots \oplus (v_1 \oplus \aleph))).$$

For example, `(foldl - 0 '(1 2 3 4))` is $(4-(3-(2-(1-0)))) = 2$. On the same list, `(foldr - 0 '(1 2 3 4))` is $(1-(2-(3-(4-0)))) = -2$.

98 ⟨*transcript* 66⟩+≡ ◁97c 100a▷

```
-> (foldl - 0 '(1 2 3 4))
2
-> (foldr - 0 '(1 2 3 4))
-2
```

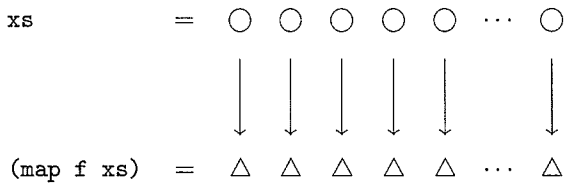
Exercises 2, 9, and 10 suggest more applications of `foldr` and `foldl`.

3.8.2 Visualizing the standard list functions

Beginning Scheme programmers are often uncertain about which list functions to use when. Using `exists?` and `all?` is relatively straightforward, but `map`, `filter`, and `foldr` can be more mysterious. This pictorial view, modeled on work by Harvey and Wright (1994), may help.

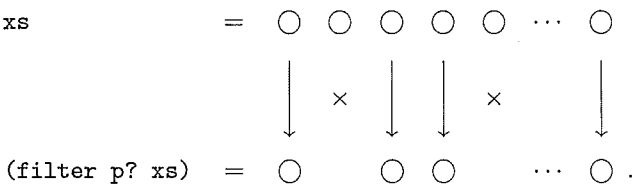
Let's suppose that a list `xs` is a list of circles:

If `f` is a function that turns circles into triangles, then `(map f xs)` turns a list of circles into a list of triangles.

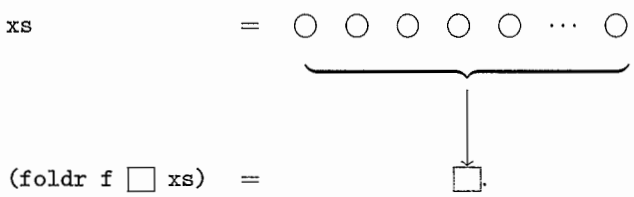


foldl 100b
foldr 100b

If `p?` is a function that takes a circle and returns true or false, then `(filter p? xs)` gives a list of just some circles:



Finally, suppose that `f` is a function that takes a circle and a box and produces another box, i.e., (`f` ○ □) = □. Then (`fold f` □ `xs`) folds all of the circles into a single box:



3.8.3 Implementing the standard list functions

Most of the higher-order list functions are easy to implement and easy to understand. They are recursive functions with one base case (the empty list) and one inductive step (`car` and `cdr`). All are part of the initial basis of μScheme.

The structure of `filter` is the same as the structure of `remove-multiples` in chunk 71d; the only difference is in the test. Filtering the empty list produces the empty list. For the induction step, we either cons or don't cons, depending on whether the `car` satisfies `p?`.

99a ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁95d 99b▷

```
(define filter (p? xs)
  (if (null? xs)
    '()
    (if (p? (car xs))
      (cons (car xs) (filter p? (cdr xs)))
      (filter p? (cdr xs)))))
```

The implementation of `map` is even simpler than that of `filter`. We don't need a conditional test in the induction step; we just apply `f` to the `car` before consing.

99b ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁99a 99c▷

```
(define map (f xs)
  (if (null? xs)
    '()
    (cons (f (car xs)) (map f (cdr xs)))))
```

Again, `exists?` is slightly different; we cut off the recursion the moment we find a satisfying element. To implement `all?`, we cut off the recursion the moment we find a *non-satisfying* element.

99c ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁99b 100b▷

```
(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)))
```

`car`	$\mathcal{P}$
`cdr`	$\mathcal{P}$
`cons`	$\mathcal{P}$
`null?`	$\mathcal{P}$

Of course, we could use De Morgan's law $\neg\forall x.P(x) = \exists x.\neg P(x)$ to define `all?` in terms of `exists?`. Negating both sides gives this definition:

100a ⟨*transcript* 66⟩+≡ ◁98 100c▷

```
-> (define alt-all? (p? xs) (not (exists? (o not p?) xs)))
-> (alt-all? even? (seq 1 10))
#f
-> (alt-all? even? '())
#t
-> (alt-all? even? (filter even? (seq 1 10)))
#t
```

Finally, the implementations of `foldr` and `foldl` are not so easy to understand. It may help to remember that `(car xs)` is always a first argument to `op`, and `zero` is always a second argument to `op`.

100b ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁99c 123b▷

```
(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))))
```

3.9 Practice IV: Higher-order functions for polymorphism

The list functions in Section 3.8 bring together functions and lists without having to know exactly what types of arguments the functions expect or what types of elements are found in the lists. For example, function `filter` can be used with function `even` to select elements of a list of numbers; or it can be used with function `(o even alist-pair-attribute)` to select elements of an association list that contains symbol-number pairs; or it can be used with infinitely many other combinations of predicates and lists. The ability to be used with arguments of many different types makes `filter` *polymorphic*. (See sidebar.) Functions `exists?`, `all?`, `map`, `foldl`, and `foldr` are also polymorphic. By contrast, a function that works with only one type of argument, like `<`, is *monomorphic*. Polymorphic functions are especially easy to reuse. In this section we study two problems we would like to solve using polymorphic functions: implementing sets and sorting lists.

car 𝒫
cdr 𝒫
cons 𝒫
curry 95d
equal? 69a
even? 96b
exists? 99c
filter 99a
not 123a
null? 𝒫
seq 71c

As shown in chunk 73a above, it's easy to implement set operations using recursive functions. But we can make the implementations more compact and (eventually) easier to understand by using the higher-order functions `exists?`, `curry`, and `foldl`.

100c ⟨*transcript* 66⟩+≡ ◁100a 102▷

```
-> (val emptyset '())
-> (define member?      (x s) (exists? ((curry equal?) x) s))
-> (define add-element  (x s) (if (member? x s) s (cons x s)))
-> (define union      (s1 s2) (foldl add-element s1 s2))
-> (define set-from-list (xs) (foldl add-element '() xs))
-> (set-from-list '(a b c x y a))
(y x c b a)
-> (union '(1 2 3 4) '(2 4 6 8))
(8 6 1 2 3 4)
```

Sidebar: Three kinds of polymorphism

Programmers love being able to reuse code, and the idea of a principled method of reuse that has a fancy Greek name is popular. So popular, in fact, that in programming languages, we can find at least three different programming techniques that are all called polymorphism.

- The sort of polymorphism we find in the standard list functions is called *parametric polymorphism.*[a] In parametric polymorphism, the polymorphic code executes the same algorithm in the same way, regardless of the types of the arguments. It is the simplest kind of polymorphism, and although it is most useful when combined with higher-order functions, implementing parametric polymorphism requires no special mechanisms at run time. Some form of parametric polymorphism is found in every functional language.

- Object-oriented languages such as Smalltalk (Chapter 10) enjoy another kind of polymorphism which is called *subtype polymorphism.* In subtype polymorphism, polymorphic code might execute *different* algorithms when operating on values of different types. For example, in a language with subtype polymorphism, squares and circles might be different types of geometric shapes, but they might both implement `draw` functions. However, the code to `draw` a circle would be different from the code to `draw` a square. Subtype polymorphism typically requires some sort of object or class system to create the subtypes. Chapter 10 is loaded with examples.

- Finally, in some languages, the same symbol can be used to stand for unrelated functions. For example, in Python, the symbol + is used not only to add numbers but also to concatenate strings. A number is not a kind of string, and a string is not a kind of number, and the algorithms are unrelated. Nonetheless, + is an operator that works on more than one type of argument, so it is considered polymorphic. This kind of polymorphism is called *ad-hoc polymorphism*, but civilized people tend to call it *overloading.*

In object-oriented parts of the world, "polymorphism" by itself usually means subtype polymorphism, and parametric polymorphism is often called *generics.* But to a dyed-in-the-wool functional programmer, *generic programming* means writing recursive functions that consume *types.*[b] If you stick with one crowd or the other, you'll quickly learn the local lingo, but if your interests become eclectic, you'll want to watch out for that word "generic."

[a] It's easy to think that the word "parametric" comes from a function's parameter or from passing functions as parameters. The word actually comes from the idea of a *type parameter*, which we define formally, as part of the language Typed μScheme, in Chapter 6.

[b] You are not expected to understand this.

These set functions work well for sets of atoms, and because `member?` calls `equal?`, they also work for sets of lists. But if we want other kinds of sets, like sets of sets or sets of association lists, these functions won't work: `equal?` is too pessimistic.

Let's focus on making sets of association lists. Two a-lists are considered equal if they have all the same keys and attributes, *regardless of how the key-attribute pairs are ordered.* In other words, although an association list is *represented* in the world of code as a *sequence* of key-attribute pairs, it should be thought of in the world of ideas as a *set* of key-value pairs. To check if two a-lists are equal, we check if each contains all the key-value pairs found in the other.

102 ⟨*transcript* 66⟩+≡ ◁100c 104a▷

```
-> (define sub-alist? (al1 al2)
     ; all of al1's pairs are found in al2
     (all? (lambda (pair)
             (equal? (alist-pair-attribute pair)
                     (find (alist-pair-key pair) al2)))
           al1))
-> (define =alist? (al1 al2)
     (if (sub-alist? al1 al2) (sub-alist? al2 al1) #f))
-> (=alist? '() '())
#t
-> (=alist? '((E coli)(I Magnin)(U Thant))
            '((E coli)(I Ching)(U Thant)))
#f
-> (=alist? '((U Thant)(I Ching)(E coli))
            '((E coli)(I Ching)(U Thant)))
#t
```

Given function `=alist?`, how should we define the set operations? One possibility is to write a new version of `member?` which uses `=alist?` instead of `equal?`. That function could be called `al-member?`. Because `add-element` calls `member?` we also have to write a new version of `add-element` which uses `al-member?` instead of `member?`. That function could be called `al-add-element`. I hope you see where this is going: we would wind up with new versions of everything except `emptyset`. Worse, if we then want to implement sets of sets, we have to do it all again. Our intentions might be good, but if we travel this road, we arrive at maintainer's hell: we have several different implementations of sets, all containing nearly identical code and all broken in the same way. For example, this implementation of sets doesn't perform very well—and if we want better-performing sets, we have to reimplement the performance improvement N times. There's a better way: instead of collecting monomorphic implementations of sets, we can use higher-order functions to write one implementation that's polymorphic.

alist-pair-attribute 74a
alist-pair-key 74a
all? 99c
equal? 69a
find 74b

3.9.1 Approaches to polymorphism in Scheme

In Scheme, polymorphism can be implemented in several ways. All the ways involve making the implementation of sets higher-order, so that instead of having `equal?` built in, the set functions rely on an `equal?` that is stored in a data structure or passed as a parameter. Depending on exactly where `equal?` is stored, we get variations:

- The simplest approach is to add a new parameter, the equality predicate, to *every* function:

103a ⟨*polymorphic-set transcript* 103a⟩≡ (603b) 103b▷

```
-> (define member? (x s eqfun)
      (exists? ((curry eqfun) x) s))
member?
-> (define add-element (x s eqfun)
      (if (member? x s eqfun) s (cons x s)))
add-element
```

and so on. Uses of the set operations would look, for example, like `(member? x s equal?)` or `(add-element x s =alist?)`.

- Adding an argument to every function is awkward. Code that uses sets must now keep track not only of the set but also the equality predicate. And the client code must be sure the same equality predicate is always passed with the same set, consistently, every time. A better plan is to *store the equality predicate with the elements*, making both part of the representation of the set. In this variation, a set is represented by a pair consisting of an equality predicate and a list of elements (with no repetitions).

103b ⟨*polymorphic-set transcript* 103a⟩+≡ (603b) ◁103a 103c▷

```
-> (define mk-set (eqfun elements) (cons eqfun elements))
-> (define eqfun-of    (set) (car set))
-> (define elements-of (set) (cdr set))
-> (val emptyset (lambda (eqfun) (mk-set eqfun '())))
emptyset
-> (define member? (x s)
      (exists? ((curry (eqfun-of s)) x) (elements-of s)))
member?
-> (define add-element (x s)
      (if (member? x s) s (mk-set (eqfun-of s) (cons x (elements-of s)))))
add-element
```

For sets of association lists, these operations are used as follows:

103c ⟨*polymorphic-set transcript* 103a⟩+≡ (603b) ◁103b

```
-> (val alist-empty (emptyset =alist?))
-> (val s (add-element '((U Thant)(I Ching)(E coli)) alist-empty))
(<procedure> ((U Thant) (I Ching) (E coli)))
-> (val s (add-element '((Hello Dolly)(Goodnight Irene)) s))
(<procedure> ((Hello Dolly) (Goodnight Irene)) ((U Thant) (I Ching) (E coli)))
-> (val s (add-element '((E coli)(I Ching)(U Thant)) s))
(<procedure> ((Hello Dolly) (Goodnight Irene)) ((U Thant) (I Ching) (E coli)))
-> (member? '((Goodnight Irene)(Hello Dolly)) s)
#t
```

The key benefit is that `emptyset` is the only set function that requires an extra parameter. The other set operations don't mention the equality function.

=alist? 102
car $\mathcal{P}$
cdr $\mathcal{P}$
cons $\mathcal{P}$
curry 95d
emptyset 73a
exists? 99c

- If there will be only a few different types of sets, but many sets of each type, then the previous solution has the disadvantage that *each set* must contain the equality function, requiring one extra cons cell per set. Moreover, the set operations require extra `car`s and `cdr`s to pull the equality function and the elements out of the representation.

  We can do better by making the equality function *part of the operations*. In particular, because each operation is a closure, we can make sure the equality function appears in the environment of that closure. The memory cost becomes independent of the number of sets that are created. Access to the elements costs nothing, and in a compiled implementation, the cost of fetching the equality function out of the closure is likely to be the same as the cost of fetching it out of a cons cell.

  To get the equality function into a closure, we define `mk-set-ops`, whose argument is an equality function, and whose result is a list of set operations:

104a ⟨*transcript* 66⟩+≡ ◁102 104b▷

```
-> (val mk-set-ops (lambda (eqfun)
      (list2
         (lambda (x s) (exists? ((curry eqfun) x) s)) ; member?
         (lambda (x s) ; add-element
               (if (exists? ((curry eqfun) x) s) s (cons x s))))))
```

  To get operations for a set of a-lists, write:

104b ⟨*transcript* 66⟩+≡ ◁104a 104c▷

```
-> (val list-of-al-ops (mk-set-ops =alist?))
-> (val al-member?     (car list-of-al-ops))
-> (val al-add-element (cadr list-of-al-ops))
```

  and so on. The operations themselves can be used without mentioning the equality function.

104c ⟨*transcript* 66⟩+≡ ◁104b 105a▷

```
-> (val emptyset '())
-> (val s (al-add-element '((U Thant)(I Ching)(E coli)) emptyset))
(((U Thant) (I Ching) (E coli)))
-> (val s (al-add-element '((Hello Dolly)(Goodnight Irene)) s))
(((Hello Dolly) (Goodnight Irene)) ((U Thant) (I Ching) (E coli)))
-> (val s (al-add-element '((E coli)(I Ching)(U Thant)) s))
(((Hello Dolly) (Goodnight Irene)) ((U Thant) (I Ching) (E coli)))
-> (al-member? '((Goodnight Irene)(Hello Dolly)) s)
#t
```


`=alist?`	102
`cadr`	125a
`car`	$\mathcal{P}$
`cons`	$\mathcal{P}$
`curry`	95d
`emptyset`	73a
`exists?`	99c
`list2`	125b

3.9.2 A polymorphic, higher-order sort

Sorting is another example of a polymorphic computation. Industrial-strength sorts are highly tuned for performance, and if we're tuning something for performance, we'd like to tune it once and use the tuned code again and again. The sorting problem resembles the set problem: to implement sets, we need an equality function that operates on set elements; and to implement sorting, we need a comparison function that operates on list elements. To sort a list of numbers, the primitive comparison function < probably suffices, but to sort a list of lists, using < makes no sense. We can define a polymorphic, higher-order function `mk-insertion-sort`, which when passed a comparison function `lt`, returns a function that sorts a list of elements into nondecreasing order according to function `lt`. The sort itself is based on the algorithm in chunk 72b.

105a ⟨*transcript* 66⟩+≡ ◁104c 105b▷

```
-> (define mk-insertion-sort (lt)
     (letrec (
       (insert (lambda (x xs)
                 (if (null? xs)
                     (list1 x)
                     (if (lt x (car xs))
                         (cons x xs)
                         (cons (car xs) (insert x (cdr xs)))))))
       (sort   (lambda (xs)
                 (if (null? xs)
                     '()
                     (insert (car xs) (sort (cdr xs)))))))
       sort))
```

The great thing about `mk-insertion-sort` is that it is easy to reuse. For example, by passing appropriate comparison functions, we get both increasing and decreasing sorts.

105b ⟨*transcript* 66⟩+≡ ◁105a 105c▷

```
-> (val sort-increasing (mk-insertion-sort <))
-> (val sort-decreasing (mk-insertion-sort >))
-> (sort-increasing '(6 9 1 7 4 3 8 5 2 10))
(1 2 3 4 5 6 7 8 9 10)
-> (sort-decreasing '(6 9 1 7 4 3 8 5 2 10))
(10 9 8 7 6 5 4 3 2 1)
```

We can also use `mk-insertion-sort` to sort pairs of integers lexicographically.

105c ⟨*transcript* 66⟩+≡ ◁105b 106b▷

```
-> (define pair< (p1 p2)
     (or (< (car p1) (car p2))
         (and (= (car p1) (car p2))
              (< (cadr p1) (cadr p2)))))
-> ((mk-insertion-sort pair<) '((4 5) (2 9) (3 3) (8 1) (2 7)))
((2 7) (2 9) (3 3) (4 5) (8 1))
```


and	123a
cadr	125a
car	$\mathcal{P}$
cdr	$\mathcal{P}$
cons	$\mathcal{P}$
list1	125b
null?	$\mathcal{P}$
or	123a

3.10 Practice V: Continuation-passing style

The sections above explore many examples of higher-order functions and their use. Most of these examples use `lambda` to capture *data* in the closure. Such data include counters, predicates, equality tests, comparison functions, and more general functions. But using these functions as data doesn't really affect the *control flow* of our computations: for example, no matter what comparison function is passed to insertion sort, the insertion-sort code continues to execute in its same way. In this section, we use higher-order functions for control flow. The key technique is to use a function call in much the same way that a C programmer or an assembly-language programmer might use a `goto`: to transfer control and never come back. A function used in this way is called a *continuation*. And a continuation is even more powerful than a `goto`, because a continuation can take arguments![11]

To develop a very simple example where a continuation would be useful, let's return to association lists. Our implementation of `find` has a little problem: we cannot distinguish between a key that is bound to *nil* and a key that is not bound at all. We could solve this problem by returning a pair: one element telling whether we found anything and another telling what it was we found. But this interface would be awkward, and using it would force us to test results twice: inside of `find` and in clients of `find`. A better solution is to pass two *continuations* to the find routine. Each continuation is a function that tells the find routine what to do next (or "how to continue") in a particular case; there is one continuation for success and one for failure. Let's see how it works:

106a ⟨*definition of* `find-c` 106a⟩≡ (106b 142b)

```
(define find-c (key alist success-cont failure-cont)
   (letrec
       ((search (lambda (alist)
                   (if (null? alist)
                       (failure-cont)
                       (if (equal? key (alist-first-key alist))
                           (success-cont (alist-first-attribute alist))
                           (search (cdr alist)))))))
     (search alist)))
```

When you call `(find-c key al succ fail)`, one of two things happens. If `key` is found in `al`, then `find-c` calls `succ` with the associated value. If `key` is not found, `find-c` calls `fail` with no arguments.

```
alist-first-attribute 74a
alist-first-key 74a
cdr P
equal? 69a
list2 125b
null? P
```

⟨*transcript* 66⟩+≡ ◁105c 107a▷

```
-> ⟨definition of find-c 106a⟩
-> (find-c 'Hello '((Hello Dolly) (Goodnight Irene))
           (lambda (v) (list2 'the-answer-is v))
           (lambda ()  'the-key-was-not-found))
(the-answer-is Dolly)
-> (find-c 'Goodbye '((Hello Dolly) (Goodnight Irene))
           (lambda (v) (list2 'the-answer-is v))
           (lambda ()  'the-key-was-not-found))
the-key-was-not-found
```

[11] Your teachers probably all hated `goto`. They probably considered it harmful. Or maybe they didn't even *tell* you about `goto`—trying to find `goto` in an introductory programming book can be like trying to find Trotsky in a picture of early Soviet leaders. If you've been unfairly deprived of `goto`, Don Knuth (1974) will remedy the injustice.

A more serious way to use `find-c` is to build a table with a default element.

107a ⟨*transcript* 66⟩+≡ ◁106b 107b▷

```
-> (define find-default (key table default)
     (find-c key table (lambda (x) x) (lambda () default)))
```

For example, to count frequencies of words, we might use a table in which the default element is zero. Function `freq` finds the frequencies of all the words in a list and lists the most frequent first. Because `freq` can be implemented by visiting each element of the list and adding one to its count, we implement `freq` using `foldr`. The `foldr` is passed `add`, which looks up a word's count in the table (default 0) and increases the count by 1.

107b ⟨*transcript* 66⟩+≡ ◁107a 107c▷

```
-> (define freq (words)
     (let
         ((add (lambda (word table)
                  (bind word (+ 1 (find-default word table 0)) table)))
          (sort (mk-insertion-sort (lambda (p1 p2) (> (cadr p1) (cadr p2))))))
       (sort (foldr add '() words))))
-> (freq '(it was the best of times , it was the worst of times ! ))
((it 2) (was 2) (the 2) (of 2) (times 2) (best 1) (, 1) (worst 1) (! 1))
```

Another problem is to find what words follow another word. (statistics about sequences of words are sometimes used to identify authorship.) To solve this problem we build a table mapping each word to the set of words that follow it; the default element is the empty set. This implementation is a bit more complicated: unlike the frequency counter, the follow-set function operates on more than one word at a time, so using `foldr` would be a little too tricky. Instead, here's an implementation that defines an auxiliary recursive function `walk`:

107c ⟨*transcript* 66⟩+≡ ◁107b 107d▷

```
-> (define followers (words)
     (letrec
         ((add (lambda (word follower table)
                  (bind word
                        (add-element follower (find-default word table '()))
                        table)))
          (walk (lambda (first rest table)
                  (if (null? rest)
                      table
                      (walk (car rest) (cdr rest) (add first (car rest) table))))))
       (walk (car words) (cdr words) '())))
-> (followers '(it was the best of times , it was the worst of times ! ))
((it (was)) (was (the)) (the (worst best)) (best (of)) (of (times)) (times (! ,)) (, length 68b of)))
```


add-element 100c
bind 74b
cadr 125a
car $\mathcal{P}$
cdr $\mathcal{P}$
curry 95d
filter 99a
find-c 106a
foldr 100b
length 68b
mk-insertion-sort 105a
null? $\mathcal{P}$

The answer doesn't even fit on the page, so let's try showing less information. Suppose we show only words that are followed by more than one word. It's easy enough to do; we use the curried form of `filter` to select elements that have more than one word in their `cadr`, then compose it with `followers`.

107d ⟨*transcript* 66⟩+≡ ◁107c 110▷

```
-> (val multi-followers
     (o
     ((curry filter) (lambda (p) (> (length (cadr p)) 1)))
     followers))
-> (multi-followers '(it was the best of times , it was the worst of times ! ))
((the (worst best)) (times (! ,)))
-> (multi-followers '(now is the time for all good men to come to the aid of the party))
((the (party aid time)) (to (the come)))
```

3.10.1 Continuation-passing for backtracking

Functions `find-c` and `find-default` are just warmups: there's no really interesting control flow going on. Below we turn our attention to a search problem with backtracking. Continuations, together with purely functional data structures, make these kinds of problems easy to solve.

The *Boolean satisfaction* problem is to find an assignment to a collection of variables such that a Boolean formula is satisfied. This problem has many industrial applications, often in verifying the correctness of a hardware or software system. For example, people who build signals for railroads want to guarantee that if the engineers obey the signals, no two trains can ever be on the same track at the same time. Surprisingly, the guarantee can be checked by solving the satisfaction problem over a very large Boolean formula. Boolean satisfaction is also widely used to check the correctness of computer hardware.

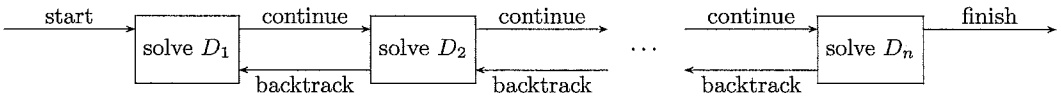
In this example, we deal only with formulas in *conjunctive normal form*; if you want to solve general Boolean formulas, do Exercise 22. A conjunctive normal form is a conjunction of disjunctions of literals; here is a grammar:

CNF	$::= D_1 \wedge D_2 \wedge \cdots \wedge D_n$	conjunction
D	$::= l_1 \vee l_2 \vee \cdots \vee l_m$	disjunction
l	$::= x \mid \neg x$	literal

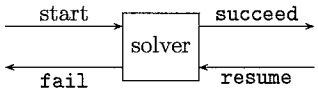
Any variable may be used in place of x. A literal x is satisfied if x is true; a literal $\neg x$ is satisfied if x is false. A disjunction is satisfied if any of is literals l_i are satisfied, and a CNF formula is satisfied if *all* of its disjunctions D_i are satisfied. Even in this simplified form, the satisfaction problem is NP-hard, which implies that in the worst case, even the best known algorithms require an exponential search. A simple formula like $x \vee y \vee z$ may have more than one satisfying assignment. In fact, 7 of the 8 possible assignments to x, y, z satisfy this formula.

Our search algorithm works with *incomplete assignments*. An incomplete assignment associates values with some variables—maybe all of them, maybe none of them, maybe some number in between. An incomplete assignment is represented by an association list in which each key is the name of a variable and each value is `#t` or `#f`. A variable that doesn't appear in the list is *unassigned*. A CNF formula is a sequence of disjunctions, and our algorithm visits the disjunctions in the sequence one at a time. Given disjunction D_i and an incomplete assignment `cur`, our algorithm tries to *extend* `cur` by adding variables in such a way that D_i is satisfied. If that works, the algorithm continues by trying to extend the assignment to satisfy D_{i+1}. But if it can't satisfy D_i, the algorithm doesn't give up; instead it *backtracks* to D_{i-1}. Maybe there is a *different* way to satisfy D_{i-1} such that it becomes possible to satisfy D_i. (In the worst case, repeatedly backtracking and trying different assignments takes exponential time.) As an example, suppose $D_1 = x \vee y \vee z$ and we choose the assignment $\{x \mapsto \texttt{\#t}, y \mapsto \texttt{\#f}, z \mapsto \texttt{\#f}\}$ to satisfy it. If $D_2 = \neg x \vee y \vee z$, then we have to go back and find a different assignment to satisfy D_1.

We can view the search as a process of going back and forth over the D_i's, tracing some path through this graph:



We implement the graph by composing individual units that look like this:



The key to keeping the implementation simple is that when we're solving an individual D_i, the disjunction solver doesn't need to keep track of what it's supposed to do next. Instead, that information is stored in two continuations, `succeed` and `fail`. In any one solver, there is a third continuation in play: if we succeed in solving D_i, we have to tell the solver for D_{i+1} how to continue if things go badly. In the solver for D_{i+1}, this continuation is just the `fail` continuation, but in the solver for D_i, it is called `resume`. Here are the details:

- To start solving, call the function `make-cnf-true` passing an empty incomplete assignment `cur` and two continuations `succeed` and `fail`. The formula being solved is a list of `disjunctions`. The first disjunction is D_1, and the call to `make-cnf-true` is represented above by the arrow labeled "start."
- If `cur` cannot possibly satisfy the first disjunction D_i, call `fail` with no parameters.
- If `cur` can satisfy D_i, or if it can be extended to satisfy D_i, then compute the (possibly extended) satisfying solution, and pass it to `succeed`. But that is not all that is passed to `succeed`.
- Because D_i might have *more than one* solution, also pass `succeed` the continuation `resume`, which it can call if it needs to backtrack and try an additional solution.

We use a similar plan to solve each of the individual D's and l's.

Before we can write code, we need to decide on a representation of formulas. Because the formulas are in conjunctive normal form, we don't need any kind of syntactic marker for the symbols $\wedge$ or $\vee$. We do need a syntactic marker for $\neg$, and we use the Scheme symbol `not`. Using the same kinds of equations we use to define binary trees in Section 3.6, we define this representation of formulas in conjunctive normal form:

$$\begin{aligned} CNF &= LIST(D) \\ D &= LIST(LIT_{\mathcal{B}}) \\ LIT_{\mathcal{B}} &= SYM \cup \{\, \texttt{(list2 'not } x\texttt{)} \mid x \in SYM \,\} \end{aligned}$$

Our code consumes S-expressions from the set *CNF*. From the structure of this set, we can plan the structure of our code. The top-level function, `make-cnf-true`, should consume one value, called `disjunctions`, from *CNF*. Because $CNF = LIST(D)$, `make-cnf-true` should be a recursive function that tests `disjunctions` with `null?` and calls itself recursively with the `cdr`. Because `(car disjunctions)` is also a list, it should be consumed by another, auxiliary recursive function, which we call `make-disjunction-true`.

110 ⟨*transcript* 66⟩+≡ ◁107d 112b▷

```
⟨Boolean solver up to and including disjunction 111a⟩
-> (define make-cnf-true (disjunctions cur fail succeed)
     (if (null? disjunctions)
       (succeed cur fail)
       (make-disjunction-true (car disjunctions) cur fail
          (lambda (cur resume)
             (make-cnf-true (cdr disjunctions) cur resume succeed)))))
```

To understand the code in more detail, look at the cases:

- If there are no more disjunctions, then `cur`, the solution so far, is a solution to the entire problem, and we pass it to `succeed`. Moreover, if `succeed` wants to "resume" to get additional solutions, there is nothing more we can do, so we pass `fail` to `succeed` as its resumption continuation.
- If there are disjunctions remaining, we try the first one using `make-disjunction-true`. If `make-disjunction-true` fails, we can't proceed, so the failure continuation for `make-disjunction-true` is `fail`.
- If `make-disjunction-true` succeeds, we continue solving the remaining disjunctions. To tell `make-disjunction-true` what to do in case of success, we create a new continuation. This continuation is the anonymous `lambda` in the body of `make-cnf-true`, which solves the remaining disjunctions by recursively calling `make-cnf-true`. If the recursive call succeeds, the whole search succeeds, so the proper success continuation for the inner recursive call is `succeed`.

  What if the recursive call fails? We shouldn't give up, because there may be an alternate solution to `(car disjunctions)` that will work. The proper failure continuation for the recursive call is therefore `resume`, which is passed to the success continuation as an argument.

car P
cdr P
make-disjunction-true 111a
null? P

We figure out the structure of `make-disjunction-true` the same way. It consumes a list of literals, so it tests them with `null?` and calls itself recursively with the `cdr`. To consume the `car` we define yet another auxiliary function, `make-literal-true`. As above, the calls are connected together by passing continuations. The success and failure continuations arrange to try every literal in turn. When we run out of literals, we have run out of solutions, and we fail.[12]

111a ⟨*Boolean solver up to and including disjunction* 111a⟩≡ (110)

```
⟨Boolean solver up to and including literals 111b⟩
-> (define make-disjunction-true (literals cur fail succeed)
     (if (null? literals)
       (fail)
       (make-literal-true (car literals) cur
          (lambda () (make-disjunction-true (cdr literals) cur fail succeed))
          succeed)))
```

When we have solved a literal successfully, we have solved the entire disjunction (because any assignment satisfying any l_i also satisfies the formula $l_1 \vee \cdots \vee l_m$), so the success continuation for `make-literal-true` is simply `succeed`. The failure continuation looks at the next literal.

You might worry that if we successfully solve the first literal, we won't revisit the later literals, and we might miss some possible solutions. But this possibility is covered by the failure continuation to `make-literal-true`; if a later solution fails, it will resume `make-literal-true`, which will fail (because each literal has at most one satisfying solution), which will then call `(make-disjunction-true (cdr literals) ...)`.

To solve a literal, we need some basic machinery, which tells us whether an assignment binds a value and whether that binding satisfies the literal. An ordinary literal is a symbol, and a negated literal is a list like `(not x)`.

111b ⟨*Boolean solver up to and including literals* 111b⟩≡ (111a) 111c▷

```
-> (define variable-of (literal)
     (if (symbol? literal)
        literal
        (cadr literal)))
-> (define binds? (literal alist)
     (find-c (variable-of literal) alist (lambda (_) #t) (lambda () #f)))
```

A literal is either a bare symbol like x, which is satisfied by `#t`, or it is a two-element list like `(not` x`)`, which is satisfied by `#f`. We can avoid a case analysis by observing that in both cases, the value that satisfies a literal `lit` is equal to `(symbol? lit)`.

111c ⟨*Boolean solver up to and including literals* 111b⟩+≡ (111a) ◁111b 112a▷

```
-> (define satisfying-value (literal)
     (symbol? literal))  ; #t satisfies 'x; #f satisfies '(not x)
-> (define satisfies? (literal alist)
     (find-c (variable-of literal) alist
             (lambda (b) (= b (satisfying-value literal)))
             (lambda () #f)))
```

An unbound variable satisfies no literals.

cadr 125a
car $\mathcal{P}$
cdr $\mathcal{P}$
find-c 106a
make-literal-true 112a
null? $\mathcal{P}$
symbol? $\mathcal{P}$

[12] Another way to think about this code is that the empty disjunction is equivalent to *false*, and there is no assignment satisfying *false*.

Finally, in `make-literal-true`, we *extend* the current assignment to satisfy a literal—provided the current assignment doesn't already bind the literal's variable. There can be at most one assignment satisfying a literal, so when we succeed in satisfying a literal, we pass a resumption continuation of `fail`.

112a ⟨*Boolean solver up to and including literals* 111b⟩+≡ (111a) ◁111c

```
-> (define make-literal-true (lit cur fail succeed)
     (if (satisfies? lit cur)
        (succeed cur fail)
        (if (binds? lit cur)
           (fail)
           (succeed (bind (variable-of lit) (satisfying-value lit) cur) fail))))
```

We now have all the machinery we need. Let us solve the formula

$$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee y \vee \neg z).$$

112b ⟨*transcript* 66⟩+≡ ◁110 112c▷

```
-> (define one-solution (formula)
     (make-cnf-true formula '() (lambda () 'no-solution)
                                (lambda (cur resume) cur)))
-> (val f '((x y z) ((not x) (not y) (not z)) (x y (not z))))
-> (one-solution f)
((x #t) (y #f))
```

This formula is true if `x` is true and `y` is false; the value of `z` doesn't matter.

We can also find all the solutions of a formula, just by changing what we do with a `resume` continuation. There are 9 distinct solutions, which is too many to fit on a page, but we can answer questions about them. Is there a solution in which x and y are both false? Both true?

112c ⟨*transcript* 66⟩+≡ ◁112b 112d▷

```
-> (define all-solutions (formula)
     (make-cnf-true formula '() (lambda () '())
                                (lambda (cur resume) (al-add-element cur (resume)))))
-> (val answers (all-solutions f))
-> (length answers)
9
-> (exists? (lambda (cur) (and (= #f (find 'x cur)) (= #f (find 'y cur)))) answers)
#f
-> (exists? (lambda (cur) (and (= #t (find 'x cur)) (= #t (find 'y cur)))) answers)
#t
```

We see that x and y cannot be false at the same time.

Not all formulas have solutions. Most obviously, we can't solve $x \wedge \neg x$.

⟨*transcript* 66⟩+≡ ◁112c 164▷

```
-> (one-solution '((x) ((not x))))
no-solution
```

Exercise 22 asks you to generalize the code above to solve *any* Boolean formula, not just in a formula in conjunctive normal form.

al-add-element 104b
and 123a
bind 74b
binds? 111b
exists? 99c
find 74b
length 68b
make-cnf-true 110
satisfies? 111c
satisfying-value 111c
variable-of 111b

3.11 Syntax and values of μScheme

The preceding sections should give you a feel for what it is like to program in μScheme and for what kinds of programming techniques are enabled by first-class, higher-order functions. We move now to a precise definition of μScheme: its syntax, its values, its semantics, and its implementation.

3.11.1 Concrete syntax

We give the full syntax of μScheme here. Everything except `lambda` and the `let` family should be familiar from Impcore, except μScheme has lots more *literal*s and *primitive*s.

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

exp ::= *literal*
| *variable-name*
| `(set` *variable-name exp*`)`
| `(if` *exp exp exp*`)`
| `(while` *exp exp*`)`
| `(begin` {*exp*}`)`
| `(`*exp* {*exp*}`)`
| `(`*let-keyword* `(`{`(`*variable-name exp*`)`}`)` *exp*`)`
| `(lambda (`*formals*`)` *exp*`)`
| *primitive*

let-keyword ::= `let` | `let*` | `letrec`

formals ::= {*variable-name*}

literal ::= *integer* | `#t` | `#f` | `'`*S-exp* | `(quote` *S-exp*`)`

S-exp ::= *literal* | *symbol-name* | `(`{*S-exp*}`)`

primitive ::= `+` | `-` | `*` | `/` | `=` | `<` | `>` | `print` | `error`
| `car` | `cdr` | `cons`
| `number?` | `symbol?` | `pair?` | `null?` | `boolean?` | `procedure?`

integer ::= sequence of digits, possibly prefixed with a minus sign

**-name* ::= sequence of characters not an *integer* and not containing `(`, `)`, `;`, or whitespace

The parser converts quoted S-expressions into literal values; the evaluator handles everything else.

3.11.2 Abstract syntax

The abstract syntax is straightforward. To Impcore we add the LETX and LAMBDAX expressions, and we generalize APPLY so that we can apply arbitrary expressions, not just names.

114a ⟨*ast.t* 114a⟩≡
```
Def* = VAL     (Name name, Exp exp)
     | EXP     (Exp)
     | DEFINE  (Name name, Lambda lambda)
     | USE     (Name)

Exp* = LITERAL (Value)
     | VAR     (Name)
     | SET     (Name name, Exp exp)
     | IFX     (Exp cond, Exp true, Exp false)
     | WHILEX  (Exp cond, Exp body)
     | BEGIN   (Explist)
     | APPLY   (Exp fn, Explist actuals)
     | LETX    (Letkeyword let, Namelist nl, Explist el, Exp body)
     | LAMBDAX (Lambda)
```

Types Value and Lambda are defined below; type Letkeyword is defined here.

114b ⟨*type definitions for μScheme* 114b⟩≡ (125c) 114d▷
```
typedef enum Letkeyword { LET, LETSTAR, LETREC } Letkeyword;
typedef struct Explist  *Explist;  /* list of Exp  */
```

3.11.3 Values

Values in Scheme are numbers, symbols, Booleans, lists, closures, or primitives. In the operational semantics, we write NUMBER(n), SYMBOL(s), BOOL(b), *nil* or CONS$\langle a, b\rangle$, $\langle\!|$LAMBDA$(\langle x_1, \ldots, x_n\rangle, e), \rho\,|\!\rangle$, and PRIMITIVE($p$).

In the interpreter, we use the C code generated from the following type definition. A Lambda structure contains a list of formal parameter names and a body; a CLOSURE is a Lambda with an environment.

114c ⟨*value.t* 114c⟩≡
```
Lambda  = (Namelist formals, Exp body)
Value   = NIL
        | BOOL (int)
        | NUM  (int)
        | SYM  (Name)
        | PAIR (Value *car, Value *cdr)
        | CLOSURE   (Lambda lambda, Env env)
        | PRIMITIVE (int tag, Primitive *function)
```

114d ⟨*type definitions for μScheme* 114b⟩+≡ (125c) ◁114b 126a▷
```
typedef struct Valuelist *Valuelist;     /* list of Value */
typedef Value (Primitive)(Exp e, int tag, Valuelist vl);
```

The definition of Primitive calls for explanation, since it would be simpler to define a primitive function as one that accepts a Valuelist and returns a Value. We pass the abstract syntax Exp e so that if the primitive fails, it can issue a suitable error message. We pass the integer tag so that we can write a single C function that implements multiple primitives. For example, the arithmetic primitives are implemented by a single function, which makes it possible for them to share the code that ensures both arguments are numbers. Implementations of the primitives appear in Section 3.14.4.

3.12 Operational semantics

Our presentation of the operational semantics for μScheme highlights the differences between μScheme and Impcore. The techniques we use are similar. The salient differences are these:

- In Impcore, we use three environments, which bind functions, global variables, and local variables respectively. In μScheme, all names are bound in a single environment ρ.
- In Impcore, environments bind names to values. In μScheme, environments bind names to *locations*, and SET changes the contents of a location, not a binding in an environment. To model the contents of locations, we use an explicit store σ. A store σ is a function from locations to values.
- In Impcore, environments are never copied, so it is safe to implement rebinding by mutating an existing binding. A μScheme program can make a copy of an environment by capturing the current environment in a closure, so it is never safe to mutate a binding in an environment; the implementation must always extend the environment with a new binding.

We assume an infinite supply of fresh locations; when we need such a location we write $\ell \notin \mathrm{dom}\,\sigma$. In this chapter, we use `malloc` to create a fresh location, but if `malloc` is called continually, eventually it will fail. In Chapter 4, we show how to reuse old locations to create the illusion of an infinite supply.

A state of an abstract machine evaluating an expression e in environment ρ is $\langle e, \rho, \sigma\rangle$. The evaluation judgment $\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$ says two things:

- The result of evaluating expression e in environment ρ, when the state of the store is σ, is the value v.
- The evaluation may update the store to produce a new state σ'.

Evaluating an expression can never change the environment; at most, evaluating an expression can change the store. Therefore, on the right-hand side of an evaluation judgment, there is no need for a new environment ρ'.

If you examine the rules carefully, you can see that we never copy a store σ, and when we make a new store σ', we never reuse the old store. These properties mean that in an implementation, we can get by with a single store, and we can mutate it, instead of having to make a new store after evaluating each expression.

3.12.1 Variables and functions

We begin our presentation with `lambda`, `let`, and function application, which are the most interesting parts of the semantics. The key ideas are these:

- Variables refer to locations, and to examine or change them we must first look up their names in ρ (to find their locations), then look at or change the contents of the locations (which means looking at σ or producing a new σ').[13]
- Let-bindings introduce fresh locations, bind them to variables, and initialize them.

[13] You might wonder if two variables ever refer to the same location. If x and y refer to the same location, and we mutate x, y changes. This behavior is called "aliasing," and it makes compiler writers miserable. In μScheme, variables cannot alias (see Exercise 44).

- Function application also introduces fresh locations, which hold actual parameters. These locations are then bound to the names of the formal parameters of the functions being applied.

Variables

The single environment makes it easy to look up the value of a variable. There are two steps: $\rho(x)$ to find the location to which x corresponds, and $\sigma(\rho(x))$ to find the value in that location. In a compiled system, these two steps are implemented at different times. At compile time, the compiler decides in what location to keep x. At run time, a machine instruction gets a value from that location.

Looking up a variable doesn't change the store σ.

$$\frac{x \in \operatorname{dom}\rho \qquad \rho(x) \in \operatorname{dom}\sigma}{\langle \textsc{var}(x), \rho, \sigma\rangle \Downarrow \langle \sigma(\rho(x)), \sigma\rangle} \qquad (\textsc{Var})$$

Likewise, assignment translates the name into a location, then changes the value of that location, producing a new store.

$$\frac{x \in \operatorname{dom}\rho \qquad \rho(x) = \ell \qquad \langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \textsc{set}(x, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\}\rangle} \qquad (\textsc{Assign})$$

The threading of the store is subtle; if in the conclusion we had extended σ rather than σ', we would have lost changes to locations carried out during evaluation of e. This behavior would be hard on the programmer, who would not expect results of `set` to disappear, and it would be especially hard on the implementor, who would have to save a copy of the old σ and reinstate it after the evaluation of e. Even for small programs, copying the store would be terribly expensive, as you can see by running the code in Section 3.15.7.

Let, let*, and letrec

To evaluate a LET expression, we evaluate *all* the bound expressions, *then* bind the names to the resulting values, and finally evaluate the main expression in the resulting environment. We make no additions to the environment until all the expressions have been evaluated.

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct} \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma_n \text{ (and all distinct)} \\ \sigma_0 = \sigma \\ \langle e_1, \rho, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \langle e, \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle \end{array}}{\langle \textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \qquad (\textsc{Let})$$

The locations $\ell_1, \ldots, \ell_n$ are understood not only to be fresh, but to be mutually distinct.

By contrast, a LETSTAR expression binds the result of each evaluation into the environment immediately. The subscripts on ρ tell the story.

$$\frac{\begin{array}{c}\rho_0 = \rho \qquad \sigma_0 = \sigma \\ \langle e_1, \rho_0, \sigma_0\rangle \Downarrow \langle v_1, \sigma_0'\rangle \qquad \ell_1 \notin \operatorname{dom}\sigma_0' \qquad \rho_1 = \rho_0\{x_1 \mapsto \ell_1\} \qquad \sigma_1 = \sigma_0'\{\ell_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_{n-1}'\rangle \qquad \ell_n \notin \operatorname{dom}\sigma_{n-1}' \qquad \rho_n = \rho_{n-1}\{x_n \mapsto \ell_n\} \qquad \sigma_n = \sigma_{n-1}'\{\ell_n \mapsto v_n\} \\ \langle e, \rho_n, \sigma_n\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \text{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(LETSTAR)}$$

Finally, LETREC binds the locations into the environment before evaluating the expressions, so that references to the new names will be valid and resolve to the new locations rather than any old ones.

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma \text{ (and all distinct)} \\ x_1, \ldots, x_n \text{ all distinct} \\ \rho' = \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\} \qquad \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \ldots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(LETREC)}$$

The contents of $\ell_1, \ldots, \ell_n$ are not specified until *after* all the $e_1, \ldots, e_n$ have been evaluated; informally, the e_i's are evaluated with the x_i's bound to new, uninitialized locations. The value of every e_i should be independent of the value of every x_j, which is of course the contents of location ℓ_j. In practice, this restriction is satisfied if every e_i is a LAMBDA expression, which is the way LETREC is normally used.

Function abstraction; function application; lexically scoped closures

Functional abstraction wraps the current environment, along with a lambda expression, in a closure. LAMBDA makes a *copy* of the current environment. Because environments can be copied, they have to map names to locations, not values. Otherwise it would be impossible for two different closures to share access to the same mutable location. (One example of such sharing is the "resettable counter" in Section 3.7.1.)

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{LAMBDA}(\langle x_1, \ldots, x_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle (\!|\text{LAMBDA}(\langle x_1, \ldots, x_n\rangle, e), \rho|\!), \sigma\rangle} \quad \text{(MKCLOSURE)}$$

Function application uses *the environment in the closure* (ρ_c), extended by binding the formal parameters to *fresh* locations. These locations are initialized by the values of the actual parameters, but during the evaluation of the body e_c, the contents of those locations might change.

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma_n \text{ (and all distinct)} \\ \langle e, \rho, \sigma\rangle \Downarrow \langle (\!|\textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e_c), \rho_c|\!), \sigma_0\rangle \\ \langle e_1, \rho, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \textsc{apply}(e, e_1, \ldots, e_n), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad (\textsc{ApplyClosure})$$

The ApplyClosure rule is only a little different from the ApplyUser rule used in Impcore.

- In μScheme, the function to be applied is indicated by an arbitrary expression, which must evaluate to a closure. In Impcore, the function is indicated by a name, which is looked up to find a user-defined function. Both a μScheme closure and an Impcore user-defined function contain formal parameters and a body, but the closure also contains the all-important environment ρ_c, which binds the free variables of the closure.[14] No such environment is needed in Impcore, because all functions are defined at top level, so any free variable can necessarily be found in the global environment ξ.

- To take care of the actual parameters, we extend not the empty environment but the enviroment ρ_c stored in the closure. And like `let`-bound names, the formal parameters are bound to fresh locations, not directly to values.

  Because the evaluation of e_c is independent of the environment ρ of the calling function, a μScheme function behaves the same way no matter where it is called from.

This method of implementing first-class, nested functions is often referred to in shorthand as *lexically scoped closures*. The "closure" part you know about; "lexically scoped" means that when you're looking at a `lambda` expression, the free variables that are needed in the closure are either bound in the global environment or are bound by "lexically enclosing" `let`s and `lambda`s. This lexical scoping makes it possible to understand a `lambda` by using purely local reasoning about the lexical context in which the `lambda` appears, which is a *static* (compile-time) property. For this reason, lexical scoping is sometimes also called *static scoping*. As an example of local reasoning about a static context, a programmer (or an implementation of Scheme) who encounters an expression like `(lambda (x) (+ x y))` can discover what `y` is simply by searching "outward" for a `let` or `lambda` that binds `y`. The search stops at the enclosing definition, which is at top level; if the search reaches top level without finding a `let` binding or `lambda` binding for `y`, then `y` must be bound in the global environment by a `val` or `define`.

[14] In our implementation, the environment actually binds all variables that are in scope, not just the variables that appear free in the `lambda` expression. Such promiscuous binding could use memory unnecessarily; for details, see Exercise 25 on page 212 in Chapter 4.

Dynamic scoping: an alternative to lexical scoping Original Lisp did not use closures; instead it represented every function as a bare LAMBDA expression. This representation admits of a very simple implementation, because the implementation never has to copy an environment. Original Lisp treats LAMBDA as a value, using the following rule for function abstraction.

$$\frac{}{\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \sigma \rangle}$$

(LAMBDA for original Lisp)

For function application, the original Lisp rule is to evaluate the body of a called function in the environment of its *caller*.

$$\frac{\begin{array}{c} \ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma_n \text{ (and all distinct)} \\ \langle e, \rho, \sigma \rangle \Downarrow \langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

(APPLYLAMBDA for original Lisp)

This rule is much easier to implement, but the rule makes `lambda` much less useful; in original Lisp, higher-order functions that contain inner `lambda`s, like `o` and `curry`, just won't work.

Using the original Lisp rule, to discover the meaning of a free variable of a `lambda` expression, an implementation of Lisp has to look at the calling function, and its caller, and so on. The context in which a function is called is a dynamic property, and the original rule is a form of *dynamic scoping*. Dynamic scoping came about because McCarthy didn't realize that closures were needed; when the original behavior was called to his attention, he characterized the semantics as a bug in the Lisp interpreter. Scheme's static lexical scoping was widely seen as an improvement, and in the 1980s, lexical scoping was adopted in Common Lisp. (If you want to experiment with a language that has dynamic scoping, try Emacs Lisp.) Tradeoffs in scoping are the subject of Exercise 25.

3.12.2 Rules for other expressions

The rules for evaluating the other expressions of μScheme are very similar to the corresponding rules of Impcore.

As in Impcore, literal values evaluate to themselves without changing the store.

$$\frac{}{\langle \text{LITERAL}(v), \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle} \quad \text{(LITERAL)}$$

The rules for conditionals, loops, and sequences are as in Impcore, except that BOOL(`#f`), not 0, stands for falsehood.

$$\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \qquad v_1 \neq \text{BOOL}(\texttt{\#f}) \qquad \langle e_2, \rho, \sigma' \rangle \Downarrow \langle v_2, \sigma'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \rho, \sigma \rangle \Downarrow \langle v_2, \sigma'' \rangle} \quad \text{(IFTRUE)}$$

$$\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \qquad v_1 = \text{BOOL}(\texttt{\#f}) \qquad \langle e_3, \rho, \sigma' \rangle \Downarrow \langle v_3, \sigma'' \rangle}{\langle \text{IF}(e_1, e_2, e_3), \rho, \sigma \rangle \Downarrow \langle v_3, \sigma'' \rangle} \quad \text{(IFFALSE)}$$

$$\frac{\langle e_1, \rho, \sigma\rangle \Downarrow \langle v_1, \sigma'\rangle \qquad v_1 \neq \text{BOOL}(\texttt{\#f}) \qquad \langle e_2, \rho, \sigma'\rangle \Downarrow \langle v_2, \sigma''\rangle \qquad \langle \text{WHILE}(e_1, e_2), \rho, \sigma''\rangle \Downarrow \langle v_3, \sigma'''\rangle}{\langle \text{WHILE}(e_1, e_2), \rho, \sigma\rangle \Downarrow \langle v_3, \sigma'''\rangle} \quad \text{(WHILEITERATE)}$$

$$\frac{\langle e_1, \rho, \sigma\rangle \Downarrow \langle v_1, \sigma'\rangle \qquad v_1 = \text{BOOL}(\texttt{\#f})}{\langle \text{WHILE}(e_1, e_2), \rho, \sigma\rangle \Downarrow \langle \text{BOOL}(\texttt{\#f}), \sigma'\rangle} \quad \text{(WHILEEND)}$$

$$\frac{}{\langle \text{BEGIN}(), \rho, \sigma\rangle \Downarrow \langle \text{BOOL}(\texttt{\#f}), \sigma\rangle} \quad \text{(EMPTYBEGIN)}$$

$$\frac{\begin{array}{c}\langle e_1, \rho, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \langle e_2, \rho, \sigma_1\rangle \Downarrow \langle v_2, \sigma_2\rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle\end{array}}{\langle \text{BEGIN}(e_1, e_2, \ldots, e_n), \rho, \sigma_0\rangle \Downarrow \langle v_n, \sigma_n\rangle} \quad \text{(BEGIN)}$$

Primitives

Compared to Impcore, μScheme has many primitives. We present only a selection.

Arithmetic The rules for primitive arithmetic operations are, at heart, the same as those for Impcore. The difference is that in Impcore, all values are numbers, whereas in μScheme, numbers are represented as NUMBER(n). As in Impcore, addition is representative.

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \text{PRIMITIVE}(\texttt{+}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle \text{NUMBER}(n), \sigma_2\rangle \\ \langle e_2, \rho, \sigma_2\rangle \Downarrow \langle \text{NUMBER}(m), \sigma_3\rangle\end{array}}{\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0\rangle \Downarrow \langle \text{NUMBER}(n+m), \sigma_3\rangle} \quad \text{(APPLYADD)}$$

Equality Testing for equality is a bit tricky. We define proof rules for an $\equiv$ relation, which is the equality implemented by the primitive =.

$$\frac{m = n \text{ (identity of numbers)}}{\text{NUMBER}(n) \equiv \text{NUMBER}(m)} \quad \text{(EQNUMBER)}$$

$$\frac{s = s' \text{ (identity of symbols)}}{\text{SYMBOL}(s) \equiv \text{SYMBOL}(s')} \quad \text{(EQSYMBOL)}$$

$$\frac{b = b' \text{ (identity of Booleans)}}{\text{BOOL}(b) \equiv \text{BOOL}(b')} \quad \text{(EQBOOL)}$$

$$\frac{}{\mathit{nil} \equiv \mathit{nil}} \quad \text{(EQNIL)}$$

For any two values v and v', we write $v \not\equiv v'$ if and only if there is no proof of $v \equiv v'$. We don't bother writing formal rules for $\not\equiv$, except to note that CONS$\langle a, b\rangle \not\equiv$ CONS$\langle a, b\rangle$, i.e., cons cells never compare equal, even when they are identical.

Given rules for $\equiv$ and $\not\equiv$, it is straightforward to define the behavior of primitive equality.

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \textsc{primitive}(\texttt{=}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle v_1, \sigma_2\rangle \\ \langle e_2, \rho, \sigma_2\rangle \Downarrow \langle v_2, \sigma_3\rangle \\ v_1 \equiv v_2\end{array}}{\langle \textsc{apply}(e, e_1, e_2), \rho, \sigma_0\rangle \Downarrow \langle \textsc{bool}(\texttt{\#t}), \sigma_3\rangle} \qquad (\textsc{ApplyEqTrue})$$

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \textsc{primitive}(\texttt{=}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle v_1, \sigma_2\rangle \\ \langle e_2, \rho, \sigma_2\rangle \Downarrow \langle v_2, \sigma_3\rangle \\ v_1 \not\equiv v_2 \text{ (i.e., no proof of } v_1 \equiv v_2)\end{array}}{\langle \textsc{apply}(e, e_1, e_2), \rho, \sigma_0\rangle \Downarrow \langle \textsc{bool}(\texttt{\#f}), \sigma_3\rangle} \qquad (\textsc{ApplyEqFalse})$$

Printing As in Impcore, the operational semantics takes no formal notice of printing, so the semantics of `print` are those of the identity function.

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \textsc{primitive}(\texttt{print}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle v, \sigma_2\rangle\end{array}}{\langle \textsc{apply}(e, e_1), \rho, \sigma_0\rangle \Downarrow \langle v, \sigma_2\rangle \qquad \text{while printing } v} \qquad (\textsc{ApplyPrint})$$

List operations The primitive CONS builds a new cons cell. We represent the cons cell as a pair of *locations* holding the values of the car and cdr.

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \textsc{primitive}(\texttt{cons}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle v_1, \sigma_2\rangle \\ \langle e_2, \rho, \sigma_2\rangle \Downarrow \langle v_2, \sigma_3\rangle \\ \ell_1 \notin \operatorname{dom}\sigma_3 \qquad \ell_2 \notin \operatorname{dom}\sigma_3 \qquad \ell_1 \neq \ell_2\end{array}}{\langle \textsc{apply}(e, e_1, e_2), \rho, \sigma_0\rangle \Downarrow \langle \textsc{cons}\langle \ell_1, \ell_2\rangle, \sigma_3\{\ell_1 \mapsto v_1, \ell_2 \mapsto v_2\}\rangle} \qquad (\textsc{Cons})$$

The primitives `car` and `cdr` observe cons cells.

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \textsc{primitive}(\texttt{car}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle \textsc{cons}\langle \ell_1, \ell_2\rangle, \sigma_2\rangle\end{array}}{\langle \textsc{apply}(e, e_1), \rho, \sigma_0\rangle \Downarrow \langle \sigma_2(\ell_1), \sigma_2\rangle} \qquad (\textsc{Car})$$

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \textsc{primitive}(\texttt{cdr}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle \textsc{cons}\langle \ell_1, \ell_2\rangle, \sigma_2\rangle\end{array}}{\langle \textsc{apply}(e, e_1), \rho, \sigma_0\rangle \Downarrow \langle \sigma_2(\ell_2), \sigma_2\rangle} \qquad (\textsc{Cdr})$$

If the result of evaluating e_1 is not a CONS cell, rules CAR and CDR do not apply, and the abstract machine gets stuck. In such a case, our interpreter issues a run-time error message.

3.12.3 Rules for evaluating definitions

Evaluating a definition may involve the computation of new values, but its primary effect is to add a new binding to the environment or to change the store. We write $\langle d, \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle$ to say that the result of evaluating definition d in environment ρ with store σ is a new environment ρ' and a new store σ'.

Global variables

When a global variable x is bound to an expression, what happens depends on whether the variable is already bound in ρ. If it is bound, then VAL is equivalent to SET.

$$\frac{x \in \operatorname{dom}\rho \qquad \langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \textsc{val}(x, e), \rho, \sigma\rangle \to \langle \rho, \sigma'\{\rho(x) \mapsto v\}\rangle} \qquad \text{(DefineOldGlobal)}$$

If x is *not* already bound to a location, the expression is evaluated in an environment in which x is bound to a fresh location, then the result of evaluating the expression is assigned to that location.

$$\frac{x \notin \operatorname{dom}\rho \qquad \ell \notin \operatorname{dom}\sigma \qquad \langle \textsc{set}(x, e), \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto \text{unspecified}\}\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \textsc{val}(x, e), \rho, \sigma\rangle \to \langle \rho\{x \mapsto \ell\}, \sigma'\rangle} \qquad \text{(DefineNewGlobal)}$$

In this definition, "unspecified" effectively means that an adversary gets to look at your code and choose the least convenient value. Writing code that depends on an unspecified value is an invitation to ruin. If we can't arrange for ruin, an unchecked run-time error will do.

You may be wondering why VAL adds the binding to the environment before having anything to put into the location. It does this in order to enable definitions of recursive functions. Consider what would happen if the binding were added after evaluating the lambda expression in the following definition of factorial.

```
(val fact
  (lambda (x)
    (if (= x 0)
      1
      (* x (fact (- x 1))))))
```

If we didn't add the binding before creating the closure, then when we evaluated the body of the lambda expression, we wouldn't have a definition for `fact`, since it would be using the environment saved in the closure, which would not contain a binding for `fact`.

Top-level functions

DEFINE is syntactic sugar for a VAL binding to a LAMBDA expression.

$$\frac{\langle \textsc{val}(f, \textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e)), \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \textsc{define}(f, \langle x_1, \ldots, x_n\rangle, e), \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle} \qquad \text{(DefineFunction)}$$

Top-level expressions

A top-level expression is syntactic sugar for a binding to the global variable `it`.

$$\frac{\langle \textsc{val}(\texttt{it}, e), \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \textsc{exp}(e), \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle} \qquad \text{(EvalExp)}$$

3.13 The initial basis

Table 3.1 lists all the functions in the initial basis of μScheme. A handful of these functions are primitives. Many of the definitions of the remaining functions appear above, in chunks named ⟨*additions to the μScheme initial basis* 67⟩. This section presents the rest of the basis functions of μScheme, except a few that are given as Exercises.

Boolean functions

Here are the basic Boolean operations, which can be implemented in terms of `if`.

123a ⟨*definitions of* `and`, `or`, *and* `not`, *for the initial basis* 123a⟩≡ (69a)

```
(define and (b c) (if b  c  b))
(define or  (b c) (if b  b  c))
(define not (b)   (if b #f #t))
```

Integer functions

We add additional integer operations, all of which are defined exactly as they would be in Impcore. We begin with comparisons.

123b ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁100b 123c▷

```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

We continue with `min` and `max`.

123c ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁123b 123d▷

```
(define max (x y) (if (> x y) x y))
(define min (x y) (if (< x y) x y))
```

Finally, we add modulus, as well as greatest common divisor and least common multiple.

123d ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁123c 125a▷

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

123e ⟨*additions to the μScheme initial basis* [[**basis-answers**]] 123e⟩≡ 156b▷

```
(define min* (ns) (foldr min (car ns) (cdr ns)))
(define max* (ns) (foldr max (car ns) (cdr ns)))
(define gcd* (ns) (foldr gcd 0 ns))
(define lcm* (ns) (foldr lcm 1 ns))
```

`=`, `!=`	Equality and inequality on atoms
`equal?`	Recursive equality on S-expressions (isomorphism, not object identity)
`/`, `*`, `-`, `+`, `mod`	Integer arithmetic
`>`, `<`, `>=`, `<=`	Integer comparison
`lcm`, `gcd`, `min`, `max`	Binary operations on integers
`lcm*`, `gcd*`, `max*`, `min*`	The same operations, but taking one non-empty list of integers as argument
`not`, `and`, `or`	Basic operations on Booleans, which, unlike their counterparts in full Scheme, evaluate all their arguments
`boolean?`, `number?`, `null?`, `pair?`, `procedure?`, `symbol?`	Type predicates
`atom?`	Type predicate saying whether a value is an atom (not a procedure and not a pair)
`cons`, `car`, `cdr`	The basic list operations
`caar`, `cdar`, `cadr`, `cddr`, `caaar`, `cdaar`, `caadr`, `cdadr`, `cadar`, `cddar`, `caddr`, `cdddr`	Abbreviations for common combinations of list operations
`list1`, `list2`, `list3`, `list4`, `list5`, `list6`, `list7`, `list8`	Convenience functions for creating lists
`length`	The length of a list
`append`	The elements of one list followed by the elements of another
`revapp`	The elements of one list, reversed, followed by another
`reverse`	A list reversed
`bind`, `find`	Insertion and lookup for association lists
`filter`	Those elements of a list satisfying a predicate
`exists?`	Does any element of a list satisfy a predicate?
`all?`	Do all elements of a list satisfy a predicate?
`map`	List of results of applying a function to each element of a list
`takewhile`	The longest prefix of a list satisfying a predicate
`dropwhile`	What's not taken by `takewhile`
`foldl`, `foldr`	Elements of a list combined by an operator, which associates to left or right, respectively
`o`	Function composition
`curry`	The curried function equivalent to some binary function
`uncurry`	The binary function equivalent to some curried function
`print`	Primitive that prints one value
`error`	Primitive that aborts the computation with an error message

Table 3.1: The initial basis of μScheme

List operations

We build in compositions of `car` and `cdr`.

125a ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁123d 125b▷

```
(define caar  (sx) (car (car  sx)))
(define cdar  (sx) (cdr (car  sx)))
(define cadr  (sx) (car (cdr  sx)))
(define cddr  (sx) (cdr (cdr  sx)))
(define caaar (sx) (car (caar sx)))
(define cdaar (sx) (cdr (caar sx)))
(define caadr (sx) (car (cadr sx)))
(define cdadr (sx) (cdr (cadr sx)))
(define cadar (sx) (car (cdar sx)))
(define cddar (sx) (cdr (cdar sx)))
(define caddr (sx) (car (cddr sx)))
(define cdddr (sx) (cdr (cddr sx)))
```

We also provide functions that create lists; full Scheme provides a variadic `list` primitive, which is more convenient.

125b ⟨*additions to the μScheme initial basis* 67⟩+≡ ◁125a

```
(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)))
```

3.14 The interpreter

This section presents the code for an interpreter written in C. As in the Impcore interpreter, we break the program into modules with well-defined interfaces. We reuse the name routines, the lexer, the input readers, and the printer from the implementation of Impcore.

3.14.1 Interfaces

As in Impcore, we gather all the interfaces into a single C header file.

125c ⟨`all.h` *for μScheme* 125c⟩≡

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <setjmp.h>
#include <ctype.h>
⟨shared type definitions 28c⟩
⟨type definitions for μScheme 114b⟩
⟨shared structure definitions 588c⟩
⟨structure definitions for μScheme (generated automatically)⟩
⟨shared function prototypes 28d⟩
⟨function prototypes for μScheme 126b⟩
```

car 𝒫
cdr 𝒫
cons 𝒫

The Environment and the Store

In the operational semantics, the store σ is intended to model the machine's memory. It should come as no surprise, then, that we use C pointers (of type `Value *`) as locations and the machine's memory as the store. An environment `Env` maps names to pointers; `find`(x, ρ) returns $\rho(x)$ if $x \in \operatorname{dom}\rho$; otherwise it returns `NULL`.

126a ⟨*type definitions for μScheme* 114b⟩+≡ (125c) ◁114d
```
typedef struct Env *Env;
```

126b ⟨*function prototypes for μScheme* 126b⟩≡ (125c) 126c▷
```
Value *find(Name name, Env env);
```

The function `bindalloc` binds a name to a freshly allocated location, and it puts a value in that location. Formally, when called with store σ, `bindalloc`(x, v, ρ) chooses an $\ell \notin \operatorname{dom}\sigma$, updates the store to be $\sigma\{\ell \mapsto v\}$, and returns the extended environment $\rho\{x \mapsto \ell\}$.

126c ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁126b 126d▷
```
Env bindalloc    (Name name,   Value v,      Env env);
Env bindalloclist(Namelist nl, Valuelist vl, Env env);
```

Calling `bindalloclist`$(\langle x_1, \ldots, x_n\rangle, \langle v_1, \ldots, v_n\rangle, \rho)$ does the same job for a list of values, returning $\rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}$, where $\ell_1, \ldots, \ell_n$ are fresh locations, which `bindalloclist` initializes to values $v_1, \ldots, v_n$.

Although our interface to environments is somewhat different from the interface we used in Impcore, you are getting old wine in new bottles. In both interpreters, the underlying representation of environments uses mutable state; that is, the environment associates each name with a pointer to a location that can be assigned to. For Impcore, we used an interface that kept the locations hidden behind the scenes. For μScheme, we expose the locations. One consequence is that the interface gets simpler; `find` replaces `isvalbound` and `fetchval`.

Impcore	*μScheme*
`isvalbound`(x, ρ)	`find`$(x, \rho) \neq$ `NULL`
`fetchval`(x, ρ)	`*find`(x, ρ)

Also, we no longer mutate an existing binding by calling `bindval`; instead, we assign directly to the location returned by `find`, so `*find`$(x, \rho) = v$ replaces `bindval`(x, v, ρ), provided $x \in \operatorname{dom}\rho$.

This interface is simpler than the interface of Chapter 2. Its drawback is that we can't explain it without appealing to the semantics of mutable locations.

Allocation

The fresh locations created by `bindalloc` and `bindalloclist` come from `allocate`. Calling `allocate(v)` finds a location $\ell \notin \operatorname{dom}\sigma$, stores `v` in ℓ (thereby updating σ), and returns ℓ.

126d ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁126c 126e▷
```
Value *allocate(Value v);
```

Before the first call to `allocate`, a client must call `initallocate`. For reasons that are disclosed in Chapter 4, `initallocate` requires `envp`, a pointer to the global environment, and it sets `*envp = NULL`.

126e ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁126d 127a▷
```
void initallocate(Env *envp);
```

Finally, if an error occurs during program execution, a client must call `resetallocate`, also passing a pointer to the global environment.

127a ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁126e 127b▷
```
void resetallocate(Env *envp);
```

Chapter 4 describes allocation in detail.

Values

The representation of values appears in chunk 114c in Section 3.11.3. The value interface also exports predefined values `truev` and `falsev`, which represent `#t` and `#f`.

127b ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁127a 127c▷
```
Value truev, falsev;
```

Before executing any code that refers to `truev` or `falsev`, clients must call `initvalue`.

127c ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁127b 127d▷
```
void initvalue(void);
```

Function `istrue` takes a value and returns `1` if the value should be regarded as true (i.e., is not `#f`) and `0` otherwise.

127d ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁127c 127e▷
```
int istrue(Value v);
```

Function `unspecified` returns an unspecified value.

127e ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁127d 127f▷
```
Value unspecified(void);
```

Evaluation

As in Impcore, `evaldef` corresponds to the $\rightarrow$ relation in our operational semantics, while `eval` corresponds to the $\Downarrow$ relation. The store σ is not passed or returned explicitly; it is represented by the C store, i.e., by the contents of memory. Our C code works by modifying this store directly.

For example, `eval(`e`, `ρ`)`, when evaluated with store σ, finds a v and a σ' such that $\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$, updates the store to be σ', and returns v.

127f ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁127e 127g▷
```
Value eval   (Exp e, Env rho);
Env   evaldef(Def d, Env rho, int echo);
```

Similarly, `evaldef(`e`, `ρ`, echo)`, when evaluated with store σ, finds a ρ' and a σ' such that $\langle e, \rho, \sigma\rangle \rightarrow \langle \rho', \sigma'\rangle$, updates the store to be σ', and returns ρ'. If `echo` is nonzero, `evaldef` also prints the name or value of whatever expression is evaluated or added to ρ.

In our implementation of μScheme, `readevalprint` has side effects on an environment pointer that is passed by reference. This technique ensures that in an interactive session, errors don't destroy the results of definitions and `val` bindings that precede the error. If `readevalprint` simply returned a new environment, then when an error occurred, any partially created environment would be lost.

127g ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁127f 128a▷
```
void readevalprint(Defreader r, Env *envp, int echo);
```

Primitives

Compared to Impcore, μScheme has many primitives. The function `primenv` returns an environment binding all the primitive operations.

128a ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁127g 128b▷
```
Env primenv(void);
```

Printing

For reference, here are the printing specifications used by the μScheme interpreter. Most of these specifications are useful only for debugging the interpreter.

- `%%` Print a percent sign
- `%d` Print an integer in decimal
- `%e` Print an `Exp`
- `%E` Print an `Explist` (list of `Exp`)
- `%\` Print a `Lambda` (in ASCII, the \ character is a common proxy for λ)
- `%n` Print a `Name`
- `%N` Print a `Namelist` (list of `Name`)
- `%p` Print a `Par`
- `%P` Print a `Parlist` (list of `Par`)
- `%r` Print an `Env`
- `%s` Print a `char*` (string)
- `%t` Print a `Def`
- `%v` Print a `Value`
- `%V` Print a `Valuelist` (list of `Value`)

Here are some of the printing functions used.

128b ⟨*function prototypes for μScheme* 126b⟩+≡ (125c) ◁128a
```
void printenv    (FILE *output, va_list_box*);
void printvalue  (FILE *output, va_list_box*);
void printclosure(FILE *output, va_list_box*);
void printexp    (FILE *output, va_list_box*);
void printdef    (FILE *output, va_list_box*);
void printlambda (FILE *output, va_list_box*);
```

3.14.2 Implementation of the evaluator

128c ⟨*eval.c* 128c⟩≡ 129a▷
```
#include "all.h"
```

⟨*eval.c declarations* 130e⟩

As in Impcore, the evaluator is still a `switch`:

129a ⟨*eval.c* 128c⟩+≡ ◁128c 131a▷

```
Value eval(Exp e, Env env) {
    switch (e->alt) {
    case LITERAL:
        ⟨evaluate e->u.literal and return the result 129b⟩
    case VAR:
        ⟨evaluate e->u.var and return the result 129c⟩
    case SET:
        ⟨evaluate e->u.set and return the result 129d⟩
    case IFX:
        ⟨evaluate e->u.ifx and return the result 133a⟩
    case WHILEX:
        ⟨evaluate e->u.whilex and return the result 133b⟩
    case BEGIN:
        ⟨evaluate e->u.begin and return the result 133c⟩
    case APPLY:
        ⟨evaluate e->u.apply and return the result 130b⟩
    case LETX:
        ⟨evaluate e->u.letx and return the result 131b⟩
    case LAMBDAX:
        ⟨evaluate e->u.lambdax and return the result 130a⟩
    }
    assert(0);
    return falsev;
}
```

Literals

As in Impcore, literals evaluate to themselves.

129b ⟨*evaluate* `e->u.literal` *and return the result* 129b⟩≡ (129a 36a)

```
return e->u.literal;
```

Variables

Variable lookup and assignment are simpler than in Impcore, because we have only one rule each. We implement $\rho(x)$ by `find`(x, ρ), we implement $\sigma(\ell)$ by $*\ell$, and we update $\sigma(\ell)$ by assigning to $*\ell$.

$$\frac{x \in \mathrm{dom}\,\rho \qquad \rho(x) \in \mathrm{dom}\,\sigma}{\langle \mathrm{VAR}(x), \rho, \sigma\rangle \Downarrow \langle \sigma(\rho(x)), \sigma\rangle} \quad (\textsc{Var})$$

129c ⟨*evaluate* `e->u.var` *and return the result* 129c⟩≡ (129a 36a)

```
if (find(e->u.var, env) == NULL)
    error("variable %n not found", e->u.var);
return *find(e->u.var, env);
```

$$\frac{x \in \mathrm{dom}\,\rho \qquad \rho(x) = \ell \qquad \langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \mathrm{SET}(x, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\}\rangle} \quad (\textsc{Assign})$$

129d ⟨*evaluate* `e->u.set` *and return the result* 129d⟩≡ (129a 36a)

```
if (find(e->u.set.name, env) == NULL)
    error("set unbound variable %n", e->u.set.name);
return *find(e->u.set.name, env) = eval(e->u.set.exp, env);
```

checkoverflow 591c
error 35b
falsev 127b
find 126b

Closures and function application

Wrapping a closure is simple; we need only to check for duplicate names.

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle (\!|\textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e), \rho\,|\!), \sigma \rangle} \quad (\textsc{MkClosure})$$

130a ⟨*evaluate* `e->u.lambdax` *and return the result* 130a⟩≡ (129a)
```
⟨if e->u.lambdax.formals contains a duplicate, call error 610c⟩
return mkClosure(e->u.lambdax, env);
```

We handle application of primitives separately from application of closures.

130b ⟨*evaluate* `e->u.apply` *and return the result* 130b⟩≡ (129a 36a)
```
{
    Value     f  = eval    (e->u.apply.fn,      env);
    Valuelist vl = evallist(e->u.apply.actuals, env);

    switch (f.alt) {
    case PRIMITIVE:
        ⟨apply f.u.primitive to vl and return the result 130c⟩
    case CLOSURE:
        ⟨apply f.u.closure to vl and return the result 130d⟩
    default:
        error("%e evaluates to non-function %v in %e", e->u.apply.fn, f, e);
    }
}
```

Applying a primitive is simpler than in our Impcore interpreter because we represent primitives by function pointers and tags. The tag is passed to the function, along with the arguments (`vl`), plus the abstract syntax `e`, which is used in error messages.

130c ⟨*apply* `f.u.primitive` *to* `vl` *and return the result* 130c⟩≡ (130b)
```
return f.u.primitive.function(e, f.u.primitive.tag, vl);
```

To apply a closure, we extend the *closure's* environment (ρ_c in the operational semantics) with the bindings for the formal variables and then evaluate the body in that environment.

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma_n \text{ (and all distinct)} \\ \langle e, \rho, \sigma \rangle \Downarrow \langle (\!|\textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c\,|\!), \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle\end{array}}{\langle \textsc{apply}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\textsc{ApplyClosure})$$

130d ⟨*apply* `f.u.closure` *to* `vl` *and return the result* 130d⟩≡ (130b)
```
{
    Namelist nl = f.u.closure.lambda.formals;
    checkargc(e, lengthNL(nl), lengthVL(vl));
    return eval(f.u.closure.lambda.body,
                bindalloclist(nl, vl, f.u.closure.env));
}
```

As in Impcore's interpreter, `evallist` evaluates a list of arguments in turn, returning a list of values.

130e ⟨*eval.c declarations* 130e⟩≡ (128c)
```
static Valuelist evallist(Explist el, Env env);
```


bindalloclist 126c
checkargc 35c
env 129a
error 35b
eval 129a
evallist 131a
lengthNL A
lengthVL A
mkClosure A

131a ⟨*eval.c* 128c⟩+≡ ◁129a

```
static Valuelist evallist(Explist el, Env env) {
    if (el == NULL) {
        return NULL;
    } else {
        Value v = eval(el->hd, env);      /* enforce uScheme's order of evaluation */
        return mkVL(v, evallist(el->tl, env));
    }
}
```

Let, let*, and letrec

Each expression in the `let` family uses its internal names and expressions to create a new environment, then evaluates the body in that environment. The rules for creating the environment depend on the keyword.

131b ⟨*evaluate* `e->u.letx` *and return the result* 131b⟩≡ (129a)

```
switch (e->u.letx.let) {
case LET:
    ⟨if e->u.letx.nl contains a duplicate, complain of error in let 610d⟩
    ⟨extend env by simultaneously binding el to nl 131c⟩
    break;
case LETSTAR:
    ⟨extend env by sequentially binding el to nl 132a⟩
    break;
case LETREC:
    ⟨if e->u.letx.nl contains a duplicate, complain of error in letrec 610e⟩
    ⟨extend env by recursively binding el to nl 132b⟩
    break;
default:
    assert(0);
}
return eval(e->u.letx.body, env);
```

A LET expression evaluates the expressions to be bound, then binds them all at once. The functions `evallist` and `bindalloclist` do all the work.

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct} \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma_n \text{ (and all distinct)} \\ \sigma_0 = \sigma \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\text{LET})$$

bindalloclist 126c
env 129a
eval 129a
mkVL $\mathcal{A}$

131c ⟨*extend* `env` *by simultaneously binding* `el` *to* `nl` 131c⟩≡ (131b)

```
env = bindalloclist(e->u.letx.nl, evallist(e->u.letx.el, env), env);
```

A LETSTAR expression binds a new name as each expression is evaluated.

$$\frac{\begin{array}{c}\rho_0 = \rho \qquad \sigma_0 = \sigma \\ \langle e_1, \rho_0, \sigma_0\rangle \Downarrow \langle v_1, \sigma_0'\rangle \qquad \ell_1 \notin \operatorname{dom}\sigma_0' \qquad \rho_1 = \rho_0\{x_1 \mapsto \ell_1\} \qquad \sigma_1 = \sigma_0'\{\ell_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_{n-1}'\rangle \qquad \ell_n \notin \operatorname{dom}\sigma_{n-1}' \qquad \rho_n = \rho_{n-1}\{x_n \mapsto \ell_n\} \qquad \sigma_n = \sigma_{n-1}'\{\ell_n \mapsto v_n\} \\ \langle e, \rho_n, \sigma_n\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \text{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(LETSTAR)}$$

132a ⟨*extend* env *by sequentially binding* el *to* nl 132a⟩≡ (131b)

```
{
    Namelist nl;
    Explist el;

    for (nl = e->u.letx.nl, el = e->u.letx.el;
         nl && el;
         nl = nl->tl, el = el->tl)
        env = bindalloc(nl->hd, eval(el->hd, env), env);
    assert(nl == NULL && el == NULL);
}
```

Finally, LETREC must bind each name to a location before evaluating any of the expressions.

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma \text{ (and all distinct)} \\ x_1, \ldots, x_n \text{ all distinct} \\ \rho' = \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\} \qquad \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \ldots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(LETREC)}$$

The initial contents of the new locations are unspecified. To be faithful to the semantics, we compute all the values before storing any of them.

bindalloc 126c
env 129a
eval 129a
evallist 131a
find 126b
unspecified 127e

⟨*extend* env *by recursively binding* el *to* nl 132b⟩≡ (131b)

```
{
    Namelist nl;
    Valuelist vl;

    for (nl = e->u.letx.nl; nl; nl = nl->tl)
        env = bindalloc(nl->hd, unspecified(), env);
    vl = evallist(e->u.letx.el, env);
    for (nl = e->u.letx.nl;
         nl && vl;
         nl = nl->tl, vl = vl->tl)
        *find(nl->hd, env) = vl->hd;
}
```

Conditional, iteration, and sequence

The implementations of the control-flow operations are very much as in Impcore. We don't bother repeating the operational semantics.

133a ⟨*evaluate* e->u.ifx *and return the result* 133a⟩≡ (129a 36a)
```
if (istrue(eval(e->u.ifx.cond, env)))
    return eval(e->u.ifx.true, env);
else
    return eval(e->u.ifx.false, env);
```

133b ⟨*evaluate* e->u.whilex *and return the result* 133b⟩≡ (129a 36a)
```
while (istrue(eval(e->u.whilex.cond, env)))
    eval(e->u.whilex.body, env);
return falsev;
```

133c ⟨*evaluate* e->u.begin *and return the result* 133c⟩≡ (129a 36a)
```
{
    Explist el;
    Value v = falsev;
    for (el = e->u.begin; el; el = el->tl)
        v = eval(el->hd, env);
    return v;
}
```

3.14.3 Evaluating definitions

The function evaldef evaluates a definition, updates the store, and returns a new environment. If echo is nonzero, evaldef also prints.

133d ⟨*evaldef.c* 133d⟩≡ 133e▷
```
#include "all.h"
```

133e ⟨*evaldef.c* 133d⟩+≡ ◁133d 135b▷
```
Env evaldef(Def d, Env env, int echo) {
    switch (d->alt) {
    case VAL:
        ⟨evaluate val binding and return new environment 134a⟩
    case EXP:
        ⟨evaluate expression, store the result in it, and return new environment 134b⟩
    case DEFINE:
        ⟨evaluate function definition and return new environment 134c⟩
    case USE:
        ⟨read in a file and return new environment 135a⟩
    default:
        assert(0);
        return NULL;
    }
}
```

env 129a
eval 129a
falsev 127b
istrue 127d

According to the operational semantics, the right-hand side of a `val` binding must be evaluated in an environment in which the name `d->u.val.name` is bound. If the binding is not already present, we bind the name to an unspecified value.

$$\frac{x \in \operatorname{dom}\rho \qquad \langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \textsc{val}(x, e), \rho, \sigma\rangle \to \langle \rho, \sigma'\{\rho(x) \mapsto v\}\rangle} \quad \text{(DefineOldGlobal)}$$

$$\frac{x \notin \operatorname{dom}\rho \qquad \ell \notin \operatorname{dom}\sigma \qquad \langle \textsc{set}(x, e), \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto \text{unspecified}\}\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \textsc{val}(x, e), \rho, \sigma\rangle \to \langle \rho\{x \mapsto \ell\}, \sigma'\rangle} \quad \text{(DefineNewGlobal)}$$

134a ⟨*evaluate* `val` *binding and return new environment* 134a⟩≡ (133e)

```
{
    Value v;

    if (find(d->u.val.name, env) == NULL)
        env = bindalloc(d->u.val.name, unspecified(), env);
    v = eval(d->u.val.exp, env);
    *find(d->u.val.name, env) = v;
    if (echo) {
        if (d->u.val.exp->alt == LAMBDAX)
            print("%n\n", d->u.val.name);
        else
            print("%v\n", v);
    }
    return env;
}
```

As in Impcore, evaluating a top-level expression has the same effect on the environment as evaluating a definition of `it`, except that the interpreter always prints the value, never the name "`it`."

134b ⟨*evaluate expression, store the result in* `it`*, and return new environment* 134b⟩≡ (133e)

```
{
    Value v = eval(d->u.exp, env);
    Value *itloc = find(strtoname("it"), env);
    if (echo)
        print("%v\n", v);
    if (itloc == NULL) {
        return bindalloc(strtoname("it"), v, env);
    } else {
        *itloc = v;
        return env;
    }
}
```

bindalloc	126c
echo	133e
env	133e
eval	127f
evaldef	133e
find	126b
mkLambdax	$\mathcal{A}$
mkVal	$\mathcal{A}$
print	33f
strtoname	28d
unspecified	127e

We rewrite DEFINE to VAL.

134c ⟨*evaluate function definition and return new environment* 134c⟩≡ (133e)

```
⟨if d->u.define.lambda.formals contains a duplicate, call error 611a⟩
return evaldef(mkVal(d->u.define.name, mkLambdax(d->u.define.lambda)),
               env, echo);
```

Reading a file is as in Impcore, except that in μScheme, we cannot mutate an environment. We therefore pass `readevalprint` a *pointer* to the environment `env`, and when `readevalprint` evaluates a definition, it writes a new environment in place of the old one.

135a ⟨*read in a file and return new environment* 135a⟩≡ (133e)

```
{
    const char *filename = nametostr(d->u.use);
    FILE *fin = fopen(filename, "r");
    if (fin == NULL)
        error("cannot open file \"%s\"", filename);
    readevalprint(defreader(filereader(filename, fin), 0), &env, 1);
    fclose(fin);
    return env;
}
```

Function `readevalprint` evaluates definitions and updates the environment `*envp`.

135b ⟨*evaldef.c* 133d⟩+≡ ◁133e

```
void readevalprint(Defreader reader, Env *envp, int echo) {
    Def d;

    while ((d = readdef(reader)))
        *envp = evaldef(d, *envp, echo);
}
```

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 `error`. 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.

3.14.4 Implementations of the primitives

We associate each primitive with a unique tag, which identifies the primitive. We also associate the primitive with a function, which implements the primitive. The tags make it easy for one function to implement more than one primitive, which makes it easy for similar primitives to share code. We use the following functions to implement primitives:

`arith`	Arithmetic functions, which expect integers as arguments
`binary`	Non-arithmetic functions of two arguments
`unary`	Functions of one argument

135c ⟨*prim.c* 135c⟩≡ 136a▷

```
#include "all.h"

static Primitive arith, binary, unary;
```

`defreader`	31f
`env`	133e
`error`	35b
`fclose`	$\mathcal{B}$
`filereader`	31c
`fopen`	$\mathcal{B}$
`nametostr`	28d
`readdef`	31e

To define the primitives and create these associations, we resort to macro madness. Each primitive appears in file `prim.h` as a macro `xx(`*name*`,` *tag*`,` *function*`)`. We 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 us ensure that the enumeration type is consistent with the initialization code.

136a ⟨*prim.c* 135c⟩+≡ ◁135c 136b▷

```
enum {
  #define xx(NAME, TAG, FUNCTION) TAG,
  #include "prim.h"
  #undef xx
  UNUSED_TAG
};
```

In `primenv`, the `xx` macros create the initial environment.

136b ⟨*prim.c* 135c⟩+≡ ◁136a 136d▷

```
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.

136c ⟨*prim.h* 136c⟩≡ 138a▷

```
xx("+", PLUS,  arith)
xx("-", MINUS, arith)
xx("*", TIMES, arith)
xx("/", DIV,   arith)
xx("<", LT,    arith)
xx(">", GT,    arith)
```

Each arithmetic primitive expects two integer arguments, so we need to be able to *project* μScheme values into C integers. The projection function `projectint` takes not only a value but also an expression, so it can issue an informative error message.

arith 137b
bindalloc 126c
env 636b
error 35b
mkPrimitive 𝒜
strtoname 28d

⟨*prim.c* 135c⟩+≡ ◁136b 137a▷

```
static int projectint(Exp e, Value v) {
    if (v.alt != NUM)
        error("in %e, expected an integer, but got %v", e, v);
    return v.u.num;
}
```

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.

137a ⟨*prim.c* 135c⟩+≡ ◁136d 137b▷

```
static int divide(int n, int 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;
}
```

Given the functions above, we can write `arith` as a function that first converts its arguments to integers, then does a `switch` to determine what to do with those integers. The result is converted back to a μScheme value by either `mkNum` or `mkBool`, both of which are generated automatically from the definition of `Value` in code chunk 114c.

137b ⟨*prim.c* 135c⟩+≡ ◁137a 138b▷

```
static Value arith(Exp e, int tag, Valuelist args) {
    int n, m;

    checkargc(e, 2, lengthVL(args));
    n = projectint(e, nthVL(args, 0));
    m = projectint(e, nthVL(args, 1));

    switch (tag) {
    case PLUS:
        return mkNum(n + m);
    case MINUS:
        return mkNum(n - m);
    case TIMES:
        return mkNum(n * m);
    case DIV:
        if (m==0)
            error("division by zero");
        return mkNum(divide(n, m));
    case LT:
        return mkBool(n < m);
    case GT:
        return mkBool(n > m);
    default:
        assert(0);
        return falsev;
    }
}
```

checkargc	35c
error	35b
falsev	127b
lengthVL	$\mathcal{A}$
mkBool	$\mathcal{A}$
mkNum	$\mathcal{A}$
nthVL	$\mathcal{A}$

Other binary primitives

μScheme has two binary primitives that do not require integer arguments. They are:

138a ⟨*prim.h* 136c⟩+≡ ◁136c 139a▷
```
xx("cons", CONS, binary)
xx("=",    EQ,   binary)
```

We implement them with the function `binary`.

138b ⟨*prim.c* 135c⟩+≡ ◁137b 139b▷
```
static Value binary(Exp e, int tag, Valuelist args) {
    Value v, w;

    checkargc(e, 2, lengthVL(args));
    v = nthVL(args, 0);
    w = nthVL(args, 1);

    switch (tag) {
    case CONS:
        ⟨return cons cell containing v and w 138c⟩
    case EQ:
        ⟨return (= v w) 138d⟩
    }
    assert(0);
    return falsev; /* not reached */
}
```

Because S-expressions are a recursive type, the representation of a cons cell has to contain pointers to S-expressions, not S-expressions themselves. Every `cons` must therefore allocate fresh locations for the pointers. This behavior makes `cons` a major source of allocation in Scheme programs.

allocate 126d
bindalloc 126c
checkargc 35c
env,
in μScheme 136b
in μScheme (in GC?!) 636b
falsev 127b
lengthVL $\mathcal{A}$
mkBool $\mathcal{A}$
mkPair $\mathcal{A}$
mkPrimitive $\mathcal{A}$
nthVL $\mathcal{A}$
strtoname 28d
truev 127b

⟨*return cons cell containing* v *and* w 138c⟩≡ (138b)
```
return mkPair(allocate(v), allocate(w));
```

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 nil.

⟨*return* (= v w) 138d⟩≡ (138b)
```
if (v.alt != w.alt)
    return falsev;

switch (v.alt) {
case NUM:
    return mkBool(v.u.num  == w.u.num);
case BOOL:
    return mkBool(v.u.bool == w.u.bool);
case SYM:
    return mkBool(v.u.sym  == w.u.sym);
case NIL:
    return truev;
default:
    return falsev;
}
```

Unary primitives

Finally, here are the unary primitives.

139a ⟨*prim.h* 136c⟩+≡ ◁138a

```
xx("boolean?",   BOOLEANP,   unary)
xx("null?",      NULLP,      unary)
xx("number?",    NUMBERP,    unary)
xx("pair?",      PAIRP,      unary)
xx("procedure?", PROCEDUREP, unary)
xx("symbol?",    SYMBOLP,    unary)
xx("car",        CAR,        unary)
xx("cdr",        CDR,        unary)
xx("print",      PRINT,      unary)
xx("error",      ERROR,      unary)
```

Here are the implementations.

139b ⟨*prim.c* 135c⟩+≡ ◁138b

```
static Value unary(Exp e, int tag, Valuelist args) {
    Value v;

    checkargc(e, 1, lengthVL(args));
    v = nthVL(args, 0);
    switch (tag) {
    case NULLP:
        return mkBool(v.alt == NIL);
    case BOOLEANP:
        return mkBool(v.alt == BOOL);
    case NUMBERP:
        return mkBool(v.alt == NUM);
    case SYMBOLP:
        return mkBool(v.alt == SYM);
    case PAIRP:
        return mkBool(v.alt == PAIR);
    case PROCEDUREP:
        return mkBool(v.alt == CLOSURE || v.alt == PRIMITIVE);
    case CAR:
        if (v.alt != PAIR)
            error("car applied to non-pair %v in %e", v, e);
        return *v.u.pair.car;
    case CDR:
        if (v.alt != PAIR)
            error("cdr applied to non-pair %v in %e", v, e);
        return *v.u.pair.cdr;
    case PRINT:
        print("%v\n", v);
        return v;
    case ERROR:
        error("%v", v);
        return v;
    default:
        assert(0);
        return falsev; /* not reached */
    }
}
```


bindalloc 126c
checkargc 35c
env,
in μScheme 136b
in μScheme (in GC?!) 636b
error 35b
falsev 127b
lengthVL A
mkBool A
mkPrimitive A
nthVL A
print 33f
strtoname 28d

3.14.5 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.

140a ⟨*scheme.c* 140a⟩≡

```
#include "all.h"

int main(int argc, char *argv[]) {
    Env env;
    Defreader input;
    int doprompt;

    doprompt = (argc <= 1) || (strcmp(argv[1], "-q") != 0);

    initvalue();
    initallocate(&env);
    ⟨install printers 140b⟩
    env = primenv();
    ⟨install into env the additions to the initial basis 141a⟩

    input = defreader(filereader("standard input", stdin), doprompt);

    while (setjmp(errorjmp))
        resetallocate(&env);
    readevalprint(input, &env, 1);
    return 0;
}
```

We have many printers.

⟨*install printers* 140b⟩≡ (140a)

```
installprinter('c', printclosure);
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);
```


_setjmp B
defreader 31f
errorjmp 35b
filereader 31c
initallocate 126e
initvalue 127c
installprinter 34b
primenv 128a
printclosure 128b
printdecimal 34c
printdef 128b
printenv 128b
printexp 128b
printexplist A
printlambda 128b
printname 28e
printnamelist A
printpar 585e
printparlist A
printpercent 34c
printstring 34c
printvalue 128b
printvaluelist A
readevalprint 127g
resetallocate 127a
stdin B
strcmp B

As in the Impcore interpreter, the C representation of the initial basis is generated automatically from code in ⟨*additions to the μScheme initial basis* 67⟩.

141a ⟨*install into* env *the additions to the initial basis* 141a⟩≡ (140a)

```
{
    ⟨C representation of initial basis for μScheme (generated automatically)⟩

    if (setjmp(errorjmp))
        assert(0);  /* fail if error occurs in basis */
    readevalprint(defreader(stringreader("initial basis", basis), 0), &env, 0);
}
```

3.14.6 Implementation of memory allocation

To allocate a new location, we call `malloc`. Chapter 4 describes a much more interesting and efficient implementation of `allocate`.

141b ⟨*loc.c* 141b⟩≡ 141c▷

```
#include "all.h"

Value* allocate(Value v) {
    Value *loc = malloc(sizeof(*loc));
    assert(loc != NULL);
    *loc = v;
    return loc;
}
```

To use `malloc` requires no special initialization or resetting. The funny references to `envp` stop compilers from warning that it isn't used.

141c ⟨*loc.c* 141b⟩+≡ ◁141b

```
void initallocate(Env *envp) {
    (void)(envp);
}
void resetallocate(Env *envp) {
    (void)(envp);
}
```

3.15 Large 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 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 make more effective use of higher-order functions than is possible in C.

_setjmp 𝓑
basis 𝓐
defreader 31f
env 140a
errorjmp 35b
malloc 𝓑
readevalprint 127g
stringreader 31c

3.15.1 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 $\forall i \geq n : i \notin \mathrm{dom}\,\sigma$.

142a ⟨*eval.scm* 142a⟩≡ 142b▷
```
(val emptystore '((next 0)))
```

We make the store a global variable `sigma`.

142b ⟨*eval.scm* 142a⟩+≡ ◁142a 142c▷
```
⟨definition of find-c 106a⟩
(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.

142c ⟨*eval.scm* 142a⟩+≡ ◁142b 142d▷
```
(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.

142d ⟨*eval.scm* 142a⟩+≡ ◁142c 143a▷
```
(define bindalloc (name v env)
  (bind name (allocate v) env))
(define bindalloclist (nl vl env)
  (if (and (null? nl) (null? vl))
    env
    (bindalloclist (cdr nl) (cdr vl) (bindalloc (car nl) (car vl) env))))
```

By insisting that in the base case, both `nl` and `vl` must be *nil*, we ensure that if `nl` and `vl` have different lengths, the interpreter issues an error message and halts.

and	123a
bind	74b
car	$\mathcal{P}$
cdr	$\mathcal{P}$
error	$\mathcal{P}$
find-c	106a
list2	125b
null?	$\mathcal{P}$

3.15.2 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.

143a ⟨*eval.scm* 142a⟩+≡ ◁142d 143b▷

```
(define apply-prim (prim)
  (lambda (args)
    (if (= (length args) 1)
      (prim (car args))
      (if (= (length args) 2)
        (prim (car args) (cadr args))
        (error (list4 'all-primitives-expect-one-or-two-arguments---got (length args)
                      ': 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.

3.15.3 The initial environment and store

We can now build the initial environment. We perform a sequence of rebindings of the empty environment.

143b ⟨*eval.scm* 142a⟩+≡ ◁143a 144a▷

```
(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 'print      (apply-prim print)      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 'procedure? (apply-prim procedure?) env))
       (env (bindalloc 'pair?      (apply-prim pair?)      env)))
    env))
```

bindalloc	142d
boolean?	$\mathcal{P}$
cadr	125a
car	$\mathcal{P}$
cdr	$\mathcal{P}$
cons	$\mathcal{P}$
error	$\mathcal{P}$
length	68b
list4	125b
null?	$\mathcal{P}$
number?	$\mathcal{P}$
pair?	$\mathcal{P}$
print	$\mathcal{P}$
procedure?	$\mathcal{P}$
symbol?	$\mathcal{P}$

3.15.4 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`.

144a ⟨*eval.scm* 142a⟩+≡ ◁143b 146a▷
```
⟨auxiliary functions for evaluation 144c⟩
(define eval (env)
  (letrec
      ((ev (lambda (e) ⟨result of evaluating expression e in environment env 144b⟩))
       ⟨letrec bindings of functions used to evaluate abstract syntax 145d⟩)
    ev))
```

Symbols are variables, the locations of which must be looked up in the environment. Other atoms evaluate to themselves.[15] Lists are function applications, unless they are abstract syntax.

144b ⟨*result of evaluating expression* `e` *in environment* `env` 144b⟩≡ (144a)
```
(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 144e⟩
          ⟨evaluate first to a function, and apply it to arguments from rest 144d⟩)))))
```

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

⟨*auxiliary functions for evaluation* 144c⟩≡ (144a) 145a▷
```
(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!

⟨*evaluate* `first` *to a function, and apply it to arguments from* `rest` 144d⟩≡ (144b)
```
((ev first) (map ev rest))
```

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

⟨*evaluate* `first` *with* `rest` *as abstract syntax* 144e⟩≡ (144b)
```
(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))))))))
```


atom? 69a
binary 145b
car $\mathcal{P}$
cdr $\mathcal{P}$
curry 95d
error $\mathcal{P}$
exists? 99c
find-c 106a
list2 125b
load 142b
map 99b
meta-begin 145f
meta-if 145d
meta-lambda 145g
meta-quote 145d
meta-set 145e
meta-while 145d
symbol? $\mathcal{P}$
trinary 145c
unary 145a

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

The auxiliary functions `unary`, `binary`, and `trinary` unpack `rest` and check its length.[16]

145a ⟨*auxiliary functions for evaluation* 144c⟩+≡ (144a) ◁144c 145b▷
```
(define unary (name f rest)
  (if (= (length rest) 1)
    (f (car rest))
    (error (list5 name 'expression-needs-one-argument,-got (length rest) 'in rest))))
```

145b ⟨*auxiliary functions for evaluation* 144c⟩+≡ (144a) ◁145a 145c▷
```
(define binary (name f rest)
  (if (= (length rest) 2)
    (f (car rest) (cadr rest))
    (error (list5 name 'expression-needs-two-arguments,-got (length rest) 'in rest))))
```

145c ⟨*auxiliary functions for evaluation* 144c⟩+≡ (144a) ◁145b
```
(define trinary (name f rest)
  (if (= (length rest) 3)
    (f (car rest) (cadr rest) (caddr rest))
    (error (list5 name 'expression-needs-three-arguments,-got (length rest) 'in
                  rest))))
```

The `ast` functions themselves are straightforward, except for `lambda`. The easiest are `quote`, `if` and `while`.

145d ⟨`letrec` *bindings of functions used to evaluate abstract syntax* 145d⟩≡ (144a) 145e▷
```
(meta-quote (lambda (e) e))
(meta-if    (lambda (e1 e2 e3) (if (ev e1) (ev e2) (ev e3))))
(meta-while (lambda (cond body) (while (ev cond) (ev body))))
```

A `set` expression requires us to find the location and rebind it.

145e ⟨`letrec` *bindings of functions used to evaluate abstract syntax* 145d⟩+≡ (144a) ◁145d 145f▷
```
(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`.

145f ⟨`letrec` *bindings of functions used to evaluate abstract syntax* 145d⟩+≡ (144a) ◁145e 145g▷
```
(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.

145g ⟨`letrec` *bindings of functions used to evaluate abstract syntax* 145d⟩+≡ (144a) ◁145f
```
(meta-lambda (lambda (formals body)
               (if (all? symbol? formals)
                 (lambda (actuals)
                   ((eval (bindalloclist formals actuals env)) body))
                 (error (list2 'lambda-with-bad-formals: formals)))))
```

```
all?          99c
bindalloclist
              142d
caddr         125a
cadr          125a
car           P
env           144a
error         P
ev            144a
eval          144a
find-variable
              144c
foldl         100b
length        68b
list2         125b
list5         125b
null?         P
store         142b
symbol?       P
```

[16] In the general case, tests of the form `(= (length xs) n)` are unjustifiable. Deciding if a list has exactly one or exactly three elements should take constant time, but calling `length` takes time linear in the length of *xs*. In the particular cses shown above, the tests are justified only because if a test fails, the program halts with an error message—thus, in working code, the time for the test is bounded by a constant. Be warned, however, that the test `(= (length xs) 0)`, however tempting, is *never* justified: the proper form is `(null? xs)`.

3.15.5 Evaluating definitions

Evaluating a definition results in a new environment.

146a ⟨*eval.scm* 142a⟩+≡ ◁144a 146e▷
```
⟨functions used to evaluate definitions 146b⟩
(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 3.10 on page 106. If `x` is bound, we leave `env` alone; otherwise we extend `env` by binding `x` to *nil*. Once `x` is safely bound, we evaluate a `set` expression.

146b ⟨*functions used to evaluate definitions* 146b⟩≡ (146a) 146c▷
```
(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.

146c ⟨*functions used to evaluate definitions* 146b⟩+≡ (146a) ◁146b 146d▷
```
(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.

⟨*functions used to evaluate definitions* 146b⟩+≡ (146a) ◁146c
```
(define meta-exp (e env)
  (begin
    (print ((eval env) e))
    env))
```

binary	145b
bindalloc	142d
car	$\mathcal{P}$
cdr	$\mathcal{P}$
error	$\mathcal{P}$
eval	144a
find-c	106a
foldl	100b
list2	125b
list3	125b
pair?	$\mathcal{P}$
print	$\mathcal{P}$
symbol?	$\mathcal{P}$
trinary	145c

3.15.6 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.

146e ⟨*eval.scm* 142a⟩+≡ ◁146a 147a▷
```
(define read-eval-print (env el)
    (foldl evaldef env el))
```

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.)

147a ⟨*eval.scm* 142a⟩+≡ ◁146e

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

3.15.7 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`.

147b ⟨*evaltest.scm* 147b⟩≡ 147c▷

```
'(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 '()))
    (print (gcd 5 10))
    (gcd 17 12)))
```

These tests also exercise `meta-while` and `meta-begin`.

147c ⟨*evaltest.scm* 147b⟩+≡ ◁147b

```
'(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)))
```

primenv 143b
read-eval-print 146e

3.16 Scheme as it really is

Full Scheme both differs from and extends μScheme. The differences are minor, but there are significant extensions in the form of imperative features, macros, and first-class continuations.

3.16.1 Language differences

Through 2007, identifiers and symbols in full Scheme were not case-sensitive; for example, `'Foo` was the same as `'foo`. The Revised[6] Report on Scheme (Sperber et al. 2009) makes full Scheme use case-sensitive identifiers and symbols, like μScheme.

Full Scheme uses `define` to introduce all top-level bindings, with slightly different syntax from μScheme.

```
(define var exp)
(define (fun args) exp)
```

The `define` form can also be used within the body of a `let` or a `lambda`.

In full Scheme, function `set` is called `set!`. Full Scheme also has `set-car!` and `set-cdr!`, which mutate the contents of existing pairs. Since mutation can create cycles, this feature complicates the implementation of `list?`.

Full Scheme does not use `find` and `bind` to operate on association lists; instead it uses `assoc`. According to the Scheme report (Kelsey, Clinger, and Rees 1998), `(assoc obj alist)` finds the first pair in `alist` whose `car` field is `equal?` to `obj`, and it returns that pair. If no pair in `alist` has `obj` as its `car`, `assoc` returns `#f`, *not* the empty list. Examples:

```
(define e '((a 1) (b 2) (c 3)))
(assoc 'a e)                              ==>  (a 1)
(assoc 'b e)                              ==>  (b 2)
(assoc 'd e)                              ==>  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))   ==>  ((a))
```

Full Scheme programs can usefully use the result of `assoc` in an `if` test, and they can also use `set-cdr!` on the result of `assoc` to mutate a binding in place.

Full Scheme has a more powerful quoting mechanism than μScheme; using *quasiquotation*, one can splice computed values or lists into quoted S-expressions.

In full Scheme, functions can take a variable number of arguments. Either the function takes one formal parameter, which is the whole list of arguments, or a formal parameter separated by a trailing period is bound to any "extra" arguments. Examples include

```
((lambda (x y . zs) zs) 3 4 5 6)   => (5 6)
((lambda xs xs)         3 4 5 6)   => (3 4 5 6)
```

Many of the primitive functions and language constructs, such as `+`, `<`, `and`, `max`, etc., accept an arbitrary number of arguments—even zero.

3.16.2 Proper tail calls

Implementations of full Scheme must be *properly tail-recursive*; this essential performance requirement goes back to the original Scheme. Informally, an implementation is properly tail-recursive if every *tail call* is implemented as a sort of "goto with arguments," that is, without pushing anything on the call stack. Arbitrarily many tail calls can be pending at once, without taking any more space than one ordinary call.

Intuitively, a call is a tail call if it is the last thing a procedure does, i.e., the result of the tail call is also the result of the calling procedure. As an important special case, proper tail recursion requires that if the last thing a full Scheme procedure does is make a recursive call to itself, the implementation makes that recursive call as efficient as a `goto`.

Many tail calls are easy to identify; for example, in a C program, the last call before a `return` or before the end of a procedure is a tail call. In a C statement `return f(`*`args`*`)`, the call to `f` is a tail call. To identify all tail calls, however, we need a more precise definition.

In Scheme, a *tail call* is a procedure call that occurs in a *tail context*. Tail contexts are defined by induction over the abstract syntax of a particular `lambda` expression. You can find the full story in Kelsey, Clinger, and Rees (1998, Section 3.5), from which we have adapted this account, but here are some example rules:

- The body of the `lambda` occurs in a tail context.
- When `(if` e_1 e_2 e_3`)` occurs in a tail context, e_2 and e_3 occur in a tail context.
- When a `let`, `let*`, or `letrec` occurs in a tail context, the body of the let occurs in a tail context.
- When `(begin` e_1 e_2 `...` e_n`)` occurs in a tail context, e_n occurs in a tail context.

In the following example the only tail call is the call to `f`. None of the calls to `g` or `h` are tail calls. The reference to `x` is in a tail context, but it is not a call and thus is not a tail call.

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (if (g) (f) #f)))
```

3.16.3 Data types

μScheme is missing many primitive data types and functions that are found in full Scheme. Full Scheme supports mutable vectors (arrays), which can be written literally using the `#(...)` notation. Full Scheme also has types for characters, mutable strings, and "I/O ports." Most implementations of Scheme provide some support for records with named fields, even though the standard (Kelsey, Clinger, and Rees 1998) does not.

Full Scheme supports lazy computations with `delay` and `force`.

The Scheme report pays careful attention to both meanings and representations of numbers. Numeric types can be arranged in a tower, in which each level contains all the levels below it:

number
complex
real
rational
integer

Numbers may also be *exact* or *inexact*. Most Scheme implementations, for example, automatically do exact arithmetic on arbitrarily large integers ("bignums"). If you are curious about how bignums work, you can implement them yourself; do Exercises 31 to 33 in Chapter 10.

3.16.4 Macros

Full Scheme has a powerful macro facility. Discussion of the macro facility is well beyond the scope of this book, but two points are noteworthy.

- The macro system is *hygienic*. This means that the macro system guarantees that macro expansion and the use of macro variables does not conflict with ordinary evaluation. In particular, to paraphrase the Scheme Report,
  - If a macro transformer inserts a binding for an identifier, the identifier is effectively renamed to avoid conflicts with other identifiers.
  - If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

  These two properties are not shared by, for example, the C macro processor.
- The hygienic nature of Scheme macros makes them incredibly powerful. Features that would require new abstract syntax in most languages can easily be defined with macros in Scheme. For example, in full Scheme, `let`, `let*`, `letrec`, and `begin` are all defined in terms of `lambda`, using macros.

3.16.5 `call/cc`

Another interesting feature of full Scheme is `call-with-current-continuation`, or `call/cc`, a function that gives the user access to the interpreter's underlying continuation. A remarkable range of control structures can be implemented in terms of `call/cc`.

To understand `call/cc`, you must first understand the idea of the "current continuation," which is "whatever the evaluator is planning to do with the value of the current expression." It is a function of one argument, taking the value of the current expression to some ultimate answer.

For example, consider the expression `(+ 3 4)`. It contains three sub-expressions, `+`, `3`, and `4`, which are evaluated in order. The "current continuation" during the evaluation of each is:

`+`: "What the evaluator is planning to do" with the value of this expression, is to apply it to the values obtained from evaluating `3` and `4`. In other words, its continuation is the function `(lambda (f) (f 3 4))`.

`3`: The evaluator is going to evaluate `4` and then add its value to the value of this expression; i.e., `(lambda (x) (+ x 4))`.

`4`: Add 3 to the value of this expression; i.e., `(lambda (x) (+ 3 x))`.

`call/cc` is a function of one argument, which is itself a one-argument function. It applies this function to the current continuation. For example, suppose `f` is the function `(lambda (k) (k 5))`. Then,

```
(+ (call/cc f) 4)
```

evaluates to 9. To see this, recall that the continuation for the middle expression (`3` in the example above) is `(lambda (x) (+ x 4))`, so `(call/cc f)` is `(f (lambda (x) (+ x 4)))`, which evaluates to 9.

Again, the utility of `call/cc` is in programming abnormal control flows. For example, it is easy to implememt `setjmp` and `longjmp` using `call/cc`. It may help your intuition to think of `call/cc` as a far more powerful version of `setjmp`.

3.17 Further reading

If you want some insight into how a language is born, John McCarthy's original paper (1960) and book (1962) about Lisp are well worth reading. But some more recent treatments of Lisp are clearer and more complete; these include books by Touretzky (1984), Wilensky (1986), Winston and Horn (1984), Graham (1993), and Friedman and Felleisen (1996). For the serious Lisper, the Common Lisp manual (Steele 1984) is an invaluable reference.

Two books dealing specifically with recursion, in the context of imperative languages, are by Rohl (1984) and Roberts (1986). Also interesting is the chapter on recursion in Reingold and Reingold 1988.

Liskov and Guttag (1986) show how to use algebraic laws to create precise specifications of the behavior of abstract data types. Parts of the functional-programming community rely heavily on algebraic laws; Bird and Wadler (1988) provide a sample, including more list laws than we present in Section 3.4. The algebraic approach can also be used on imperative programs, albeit with greater difficulty; Hoare et al. (1987) present algebraic laws for a simple imperative language.

Abelson and Sussman's (1985) book on programming in Scheme is noted for its many interesting examples and difficult problems. Harvey and Wright (1994) give a more introductory treatment of Scheme, aimed at students with little programming experience. Felleisen et al. (2001) also teach introductory programming using Scheme, or rather, using five subsets of Scheme. The subsets are carefully crafted to help raw beginners evolve into successful Schemers.

Dybvig's (1987) book on Scheme is more like an expanded reference manual, but as such it is clear and well organized. Slade's book (1987) on T, a Scheme-like dialect of Lisp, is longer and has more examples. The series of articles by Steele and Sussman (Sussman and Steele 1975; Steele and Sussman 1976, 1978) introducing Scheme make entertaining reading; we especially recommend Steele and Sussman 1978. There is also a report giving the official definition of the language (Kelsey, Clinger, and Rees 1998).

The idea of using lexically scoped closures to implement first-class, nested functions did not originate with Scheme. The idea had been developed in a number of languages before Scheme, mostly in Europe. Examples include Iswim (Landin 1966), Pop-2 (Burstall, Collins, and Popplestone 1971), and Hope (Burstall, MacQueen, and Sannella 1980). The book by Henderson (1980) is from this school. Also highly recommended is the short, but very interesting, book by Burge (1975).

Reynolds (1972) shows how to write "definitional" interpreters, which is a strong motivator for the use of continuations. The continuation-based backtracking search in Section 3.10 is based on a "Byrd box," which was used to understand Prolog programs (Byrd 1980); we use the terminology of Proebsting (1997), who describes an implentation of Icon, a programming language that has backtracking built in (Griswold and Griswold 1996).

Clinger (1998) gives a formal, precise definition of proper tail recursion and explores some of the implications.

3.18 Exercises

Highlights

Here are some of the highlights of the exercises below:

- Exercise 12 on page 159 challenges you to understand the idea of "fold" well enough that you can invent your own "fold-like" higher-order function.
- Exercise 15 on page 159 asks you to implement a "data structure" whose values are represented as functions.
- Exercise 17 on page 161 guides you to an *efficient*, purely functional representation of queues, without using mutation.
- Exercise 22 on page 163 invites you to work toward mastery of continuation-passing style by implementing a solver for general Boolean formulas. This Exercise will also help you develop the habit of writing only as many auxiliary functions as you really need, and no more.
- Exercises 32 and 34 on page 165 ask you to prove some classic algebraic laws of pure functional programming: `append` is associative, and the composition of `maps` is the `map` of the composition. The proofs require you to combine equational reasoning with induction. The laws can be used to improve programs, and they are also used by optimizing compilers for languages like Haskell.

Guide to all the exercises

Exercise 1 will help you think more deeply about the ways that S-expressions can be used to represent data. These deeper thoughts will help with Exercises 2 to 4, which serve two purposes: to develop your skills in programming with S-expressions, and to get you to a point, if you are not there already, where you find it perfectly comfortable and natural to write recursive functions.

Exercise 5 gives you practice writing recursive functions on lists. And it introduces you to `takewhile` and `dropwhile`, two very reusable functions.

Exercises 6 and 7 ask you to develop simple higher-order functions for a single use only. Exercise 6 asks you to create a higher-order sorting function; Exercise 7 asks you to create a higher-order function that returns operations on binary search trees. Both exercises provide more practice writing recursive functions and will help prepare you to create more ambitious "fold-like" functions.

Exercises 8 and 9 will force you to start familiarizing yourself with some of the higher-order basis functions in Table 3.1 on page 124. These functions are deeply connected to a central idea of functional programming: higher-order functions enable us to create powerful tools which we can use to put together new computations out of existing, simple parts. Exercise 10 is similar, but it focuses exclusively on the `fold` functions, and it will help you get a feeling for why fold is sometimes called a "universal" operation.

The *idea* of "fold" generalizes to any list-like or tree-like data structure. Exercise 11 asks you to develop this idea for a preorder traversal. Exercise 12 asks the same for a dot product, but it leaves you more to figure out for yourself.

Exercise 13 asks you to think about allocation—an operation whose cost matters—and to reduce allocation using the method of accumulating parameters from Section 3.3.2.

Exercise 14 helps you check your understanding of `let`.

Exercise 15 asks you to develop code using a classic functional-programming technique: what you normally think of as *data* is actually represented as a *function*. This kind of exercise changes people's thinking. Unless you are already familiar with functional programming—and if you are, why are you reading this chapter?—this exercise is a must.

Exercise 16 continues the theme of using functions to represent data, pushing it to an extreme. Do this exercise if you want a challenge, or if at some point you're going to study *lambda calculus*, you might want to come back to this exercise.

A classic problem in every functional language is that a list is not an efficient representation of a queue: a list provides efficient access to just one end, but a queue needs efficient access to both ends. This classic problem has a classic solution, which Exercise 17 asks you to develop. This solution is also one of the classic examples of a data structure that demands an *amortized* analysis.

Exercise 18 asks to you code a classic algorithm: topological sort. This exercise requires a lot of code by Scheme standards; I wrote two solutions, both of about 100 lines. Both solutions incorporate a number of auxiliary "helper" functions. The exercise is more about polishing your programming skills than about Scheme in particular, but if you want to see how a familiar, imperative algorithm plays out in a functional settings, it could be a good one.

Exercises 19 to 21 build on the metacircular evaluator developed in Section 3.15. The exercises include some improvements to the language as well as some attention to performance.

Exercise 22 asks you to apply the continuation-passing ideas in Section 3.10. It also asks you to consider how structured data (here, formulas) might be represented as S-expressions. Continuations are one of the most important advanced ideas in programming, and they play a crucial role in the design of Smalltalk (Chapter 10), so this exercise is highly recommended.

Exercise 23 asks you to go to the source to learn more about first-class continuations.

Exercises 24 and 25 are design questions. Exercise 24 asks a narrow question: how should equality be extended to functions? Exercise 25 asks you to investigate and analyze dynamic scoping. It's a way of digging into the history of scoping rules in programming languages, and to answer it, you'll need to spend considerable time in the library.

Exercises 26 to 36 ask you to practice calculational proofs in the style of Section 3.4.5. Some functional programmers think this style of proof is central to functional languages. The first few exercises require only simple proofs. The proofs about recursive function require some mathematical induction. A couple of the results are classics, like the associativity of `append` (Exercise 32) and the composition of `maps` (Exercise 34).

Exercise 37 asks to you express a simple language-design decision in operational semantics.

Exercise 38 asks to you make a more challenging change in the language design: rule out such nonsense as `(val u u)` while still permitting the definition of recursive functions.

Exercise 39 asks to you explore the semantics in a different way: by writing a short program that behaves differently under different semantic rules, or by explaining why no such program is possible.

Exercises 41 and 42 shift your attention to the operational semantics of primitives. Exercise 41 asks you to write a semantics for mutating primitives, and Exercise 42 asks you to explore what happens if a cons cell holds values instead of locations.

Exercise 43 asks you to prove that μScheme can execute Impcore functions. The proof is ambitious. Exercise 44 asks you to prove that in μScheme, assigning to one variable can never change the value of another.

Exercise 45 asks you to build and evaluate a version of the μScheme interpreter which guarantees to reject μScheme computations that depend on unspecified values.

Exercise 46 asks you to show that in the implementation of μScheme, the size of the environment is bounded, and the bound is determined by the μScheme source code. This fact is used by compiler writers to develop very efficient representations of environments.

Exercises 47 to 49 ask you to add new primitives to μScheme. These exercises will help you get acclimated to the implementation and will also help you understand what language primitives are all about, and why it's much easier to add a new *primitive* than to add new *syntax*.

Exercise 50 asks you to add a trace facility to the μScheme interpreter. The trace facility is worth having, because it makes debugging μScheme programs much easier, but the primary value of this exercise is that it forces you to develop a deeper understanding of how `eval` works. This understanding will be especially useful if you intend to build either of the garbage collectors in Chapter 4.

Exercise 51 exposes you to the kind of issue that arises in industrial-strength interpreters: how can you be sure not to leak open file descriptors? This exercise is about perfecting your craft as an implementor, not about ideas in programming languages.

Exercise 52 walks you through the kind of experiments you will have to do if you get serious about the performance of an interpreter.

Exercise 53 asks you to extend the μScheme interpreter so that it can read Scheme dot notation in quoted S-expressions. The main lesson is probably that string-processing in C is not much fun—but it will make you appreciate the code in Chapter 5 and Appendix E.

Learning about the language

As you tackle the exercises below, you may find it useful to refer to Table 3.1 on page 124, which lists all the functions in μScheme's initial basis.

1. Prove that a list of S-expressions is also an S-expression, that is, prove that

$$LIST(SEXP) \subseteq SEXP.$$

   Hint: try proof by contradiction.

   This fact has implications for programming: if you are writing a recursive function that consumes a list of S-expressions, you also have a the choice of writing a more general function that consumes an arbitrary S-expression. Sometimes the more general function is simpler.

2. Write these functions:

   (a) When `x` is an atom and `xs` is in $LIST(SEXP)$, `(count x xs)` returns the number of (top-level) elements of `xs` that are equal to `x`. Assume `x` is not nil.

   ⟨*exercise transcripts* 154a⟩≡
   ```
   -> (count    'a '(1 b a (c a)))
   1
   ```

   (b) When `x` is an atom and `xs` is in $LIST(SEXP)$, `(countall x xs)` returns the number of times `x` occurs *anywhere* in `xs`, not only at top level. Assume `x` is not nil.

   ⟨*exercise transcripts* 154a⟩+≡
   ```
   -> (countall 'a '(1 b a (c a)))
   2
   ```

(c) When `xs` is a list of S-expressions, `(mirror xs)` returns a list in which every list in `xs` is recursively mirrored, and the resulting lists are in reverse order.

⟨*exercise transcripts* 154a⟩+≡
```
-> (mirror '(1 2 3 4 5))
(5 4 3 2 1)
-> (mirror '((a (b 5)) (c d) e))
(e (d c) ((5 b) a))
```

Informally, `mirror` returns the S-expression you would get if you looked at its argument in a vertically oriented mirror, except that the individual atoms are not reversed. (Try putting a mirror to the right of the example, facing left.) A more precise way to say what `mirror` does is that it consumes an S-expression and returns the S-expression that you would get if you wrote the parentheses and atoms of the original S-expression in reverse order, exchanging open parentheses for close parentheses and vice versa.

(d) Function `flatten` consumes a list of S-expressions and erases internal parentheses. That is, when `xs` is a list of S-expressions, `(flatten xs)` constructs a list having the same atoms as `xs` in the same order, but in a flat list. For purposes of this problem, `'()` should be considered not as an atom but as an empty list of atoms.

⟨*exercise transcripts* 154a⟩+≡
```
-> (flatten '((I Ching) (U Thant) (E Coli)))
(I Ching U Thant E Coli)
-> (flatten '((((((a))))))
(a)
-> (flatten '((a b) ((c d) e)))
(a b c d e)
```

(e) When both `xs` and `ys` are in set $LIST(ATOM)$, `(contig-sublist? xs ys)` determines whether the list `xs` is a contiguous subsequence of the list `ys`. That is, `(contig-sublist? xs ys)` returns `#t` if and only if there are two other lists `front` and `back`, such that `ys` is equal to `(append (append front xs) back)`.

⟨*exercise transcripts* 154a⟩+≡
```
-> (contig-sublist? '(a b c) '(x a y b z c))
#f
-> (contig-sublist? '(a y b) '(x a y b z c))
#t
-> (contig-sublist? '(x)     '(x a y b z c))
#t
```

(f) When both `xs` and `ys` are in set $LIST(ATOM)$, `(sublist? xs ys)` determines whether the list `xs` is a mathematical subsequence of the list `ys`. That is, `(sublist? xs ys)` returns `#t` if and only if the list `ys` contains the elements of `xs`, in the same order, but possibly with other values in between.

⟨*exercise transcripts* 154a⟩+≡
```
-> (sublist? '(a b c) '(x a y b z c))
#t
-> (sublist? '(a y b) '(x a y b z c))
#t
-> (sublist? '(a z b) '(x a y b z c))
#f
-> (sublist? '(x y z) '(x a y b z c))
#t
```

3. Write the following set functions:

   (a) `(remove x s)` returns a set having the same elements as set `s` with element `x` removed.

   (b) `(subset? s1 s2)` determines if `s1` is a subset of `s2`.

   (c) `(=set? s1 s2)` determines if lists `s1` and `s2` represent the same set.

4. This set of questions concerns tree traversal.

   (a) Using the representation defined in Section 3.6, program post-order and in-order traversal for binary trees.

   (b) Extend the pre-order and level-order traversals to trees of arbitrary degree, represented in such a way that binary trees have the same representation given them in the text. For example, `'(a b c d)` represents a ternary tree whose root is labeled with `a` and which has three children, labeled `b`, `c`, and `d`, respectively, all leaf nodes. Note that there are two ways to represent leaf nodes: `'a` and `'(a)` both represent a leaf node labeled `a`.

5. Function `takewhile` takes a predicate and a list and returns the longest prefix of the list in which every element satisfies the predicate. Function `dropwhile` removes the longest prefix and returns whatever is left over.

   ⟨*exercise transcript* 156a⟩≡
   ```
   -> (define even? (x) (= (mod x 2) 0))
   -> (takewhile even? '(2 4 6 7 8 10 12))
   (2 4 6)
   -> (dropwhile even? '(2 4 6 7 8 10 12))
   (7 8 10 12)
   ```

   Implement `takewhile` and `dropwhile`.

   (You might also look at Exercise 36 on page 165, which asks you to prove a basic law about `takewhile` and `dropwhile`.)

   ⟨*additions to the μScheme initial basis* **[[basis-answers]]** 123e⟩+≡
   ```
   (define takewhile (p? xs)
     (if (null? xs)
         '()
         (if (p? (car xs))
             (cons (car xs) (takewhile p? (cdr xs)))
             '())))
   (define dropwhile (p? xs)
     (if (null? xs)
         '()
         (if (p? (car xs))
             (dropwhile p? (cdr xs))
             xs)))
   ```

6. Chunk 105c shows `pair<`, a function that compares pairs of integers lexicographically. Write a higher-order function `lex-<*` to compare lists of differing lengths lexicographically. It should take as a parameter an ordering on the elements of the list. Be careful not to make any assumptions about the elements, other than that they can be ordered by the parameter.

⟨*exercise transcripts* 154a⟩+≡
```
-> (val alpha-order (lex-<* <))
-> (alpha-order '(4 15 7) '(4 15 7 13 1))
#t
-> (alpha-order  '(4 15 7) '(4 15 5))
#f
```

This lexicographic ordering of lists is closely related to alphabetical ordering; to see the relationship, translate the numbers to letters: the two results above say DOG < DOGMA and DOG $\not<$ DOE.

7. Write a higher-order implementation of binary-search trees. Function `mk-search` should take a comparison function as argument and return a list of 2 functions: `lookup` and `insert`.

8. Define function `remove-multiples-too`, which does what `remove-multiples` does, but works by using the higher-order functions in the initial basis. Do not use `lambda`, `if`, or recursion. You'll need to copy function `divides` from chunk 71c. All the other functions you'll need are in the initial basis.

⟨*exercise transcripts* 154a⟩+≡
```
-> (remove-multiples-too 2 '(2 3 4 5 6 7))
(3 5 7)
```

9. Use `map`, `curry`, `foldl`, and `foldr` to define the following functions:

   (a) `cdr*`, which takes the `cdr`'s of each element of a list of lists:

   ⟨*exercise transcripts* 154a⟩+≡
   ```
   -> (cdr* '((a b c) (d e) (f)))
   ((b c) (e) ())
   ```

(b) `max*`, which finds the maximum of a non-empty list of integers. Function `max` does not have a left or right identity.

(c) `gcd*`, which finds the greatest common divisor of a non-empty list of integers. As implemented in μScheme, function `gcd` has left and right identities, but you can also implement `gcd*` in the same way you implement `max*`.

(d) `lcm*`, which finds the least common multiple of a non-empty list of integers. Like `gcd`, `lcm` has left and right identities, but again you can write `lcm*` in the same way as `max*`.

(e) `sum`, which finds the sum of a non-empty list of integers.

(f) `product`, which finds the product of a non-empty list of integers.

⟨*exercise transcripts* 154a⟩+≡
```
-> (sum '(1 2 3 4))
10
-> (product '(1 2 3 4))
24
-> (max* '(1 2 3 4))
4
-> (lcm* '(1 2 3 4))
12
```

(g) `append`, which appends two lists.

(h) `snoc`, which adds its first argument (an arbitrary S-expression) as the last element of its second argument (a list):

⟨*exercise transcripts* 154a⟩+≡
```
-> (snoc 'a '(b c d))
(b c d a)
```

The word "`snoc`" is `cons` spelled backward.

(i) `reverse`, which reverses a list.

(j) `insertion-sort`, which sorts a list of numbers. You may take `insert` as given in chunk 72b.

(k) `mkpairfn`, which when applied to an argument v, returns a function that when applied to a list of lists, places v in front of each element:

⟨*exercise transcripts* 154a⟩+≡
```
-> ((mkpairfn 'a) '() (b c) (d) ((e f)))
((a) (a b c) (a d) (a (e f)))
```

It is possible to define all six functions using only one `define` and one `lambda`. Total.

10. Use `foldr` or `foldl` to implement `length`, `map`, `filter`, `exists?`, and `all?`. It is OK if some of these functions do a bit more work than their official versions, as long as they produce the same answers.

    For the best possible solution, craft your answers so that no adversary can write a μScheme program that can tell whether a function you wrote has been substituted for an official version. The adversary is limited to writing new code and may not change the meanings of names that are in the initial basis. (Hint: a μScheme program that uses `set` can tell a lot more than a μScheme program that doesn't use `set`.)

11. Function `pre-ord` in Section 3.6 is not so useful if, for example, one wants to perform some other computation on a tree, like finding its height. By analogy with `foldl` and `foldr`, define `fold-pre-ord`. In addition to the tree, it should take two arguments: a function to be applied to internal nodes, and a function to be applied to leaves. Here are some examples:

⟨*exercise transcripts* 154a⟩+≡
```
-> (fold-pre-ord (lambda (label left right) (cons label (append left right)))
                 (lambda (label) (list1 label))
       '(A (B C D) (E (F G H) I)))
(A B C D E F G H I)
-> (define tree-height (t)
     (fold-pre-ord (lambda (label left right) (+ 1 (max left right)))
                   (lambda (label) 1)
                   t))
-> (tree-height '(A (B C D) (E (F G H) I)))
4
```

12. Here is another opportunity for a fold, in two parts:

    (a) Implement function `dot-product`, which computes the dot product of two vectors, reprsented as lists. (If the vectors are $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$, the dot product is $u_1v_1 + \cdots + u_nv_n$.)

    ⟨*exercise transcripts* 154a⟩+≡
    ```
    -> (dot-product '(1 2 3) '(10 5 2))
    26
    ```

    (b) Generalize the computation into a "fold-like" function.

13. Function `pre-ord` in Section 3.6 may allocate unnecessary cons cells. Using the method of accumulating parameters from Section 3.3.2, rewrite `pre-ord` so that the number of cons cells allocated is exactly equal to the number of cons cells in the final answer.

14. What does the following procedure do? How can it possibly work?

⟨*exercise transcripts* 154a⟩+≡
```
⟨definition of sqrt 603a⟩
-> (define vector-length (x y)
     (let ((+ *)
           (* +))
       (sqrt (* (+ x x) (+ y y)))))
-> (vector-length 3 4)
5
```

append	70a
cons	$\mathcal{P}$
list1	125b
max	123c
sqrt	603a

15. An alternate representation for sets in Scheme is as Boolean-valued functions, called "characteristic functions." In particular, the null set is represented by a function that always returns `#f`, and the membership test is just application:

⟨*exercise transcripts* 154a⟩+≡
```
-> (val emptyset (lambda (x) #f))
-> (define member?  (x s) (s x))
```

(a) Write `add-element`, `union`, `inter`, and `diff` using this new representation. (The set `(diff s1 s2)` is the set that contains every element of `s1` that is not also in `s2`.)

(b) Write the third approach to polymorphism (page 104, Section 3.9.1) using this representation.

16. This exercise explores the power of `lambda`, which can do more than you might have expected.

(a) We claim above that *any* implementation of `cons`, `car`, and `cdr` is acceptable provided it satisfies the list laws in Section 3.4.1. Define `cons`, `car`, `cdr`, and `nil` using only `if`, `lambda`, function application, the primitive `=`, and the literals `#t` and `#f`.

⟨*exercise transcripts* 154a⟩+≡

```
-> (define nth (n xs)
     (if (= n 1) (car xs)
                 (nth (- n 1) (cdr xs))))
nth
-> (val ordinals (cons 'first (cons 'second (cons 'third nil))))
-> (nth 2 ordinals)
second
-> (nth 3 ordinals)
third
```

(b) Under the same restrictions, implement `null?`.

(c) Define `cons`, `car`, `cdr`, `null?`, and `nil` using only `lambda` and function application. Don't use `if`, `=`, or any literal values.

The hard part is implementing `null?`, because you can't use the standard representation of Booleans. Instead, you have to invent a new representation of truth and falsehood. This new representation won't support the standard `if`, so you have to develop an alternative, and you have to explain how to use that alternative in place of `if`. In other words, explain what you should write instead of `(if` e_1 e_2 e_3`)`. Once you have an alternative to `if` and a new representation of Booleans, you will be able to implement `null?` using only `lambda` and function application.

```
car    P
cdr    P
cons   P
```

17. The implementation of queues in Section 3.6 is simple but inefficient; `enqueue` can take time and space proportional to the number of elements in the queue. The problem is the representation: the queue operations have to get to both ends of a the queue, and getting to the back of a list requires linear time. A better representation uses *two* lists:

    - The list `front` represents the front of the queue. It stores older elements, and they are ordered with the oldest element at the beginning, so `front` and `without-front` can be implemented using `car` and `cdr`.
    - The list `back` represents the back of the queue. It stores young elements, and they are ordered with the youngest element at the beginning, so `enqueue` can be implemented using `cons`.

    The trick of this representation is that when the `front` elements are exhausted, the `back` elements must somehow be transferred to the `front`. Using this idea, implement queues efficiently.

    An *amortized analysis* can show that using this implementation of queues, each operation takes constant time on average.

18. It is easy to use a Scheme list to represent a directed graph. For example, each node can be labeled with a distinct symbol, and the graph can be represented as a list of edges, where each edge is a list containing the labels of that edge's source and destination. Write a function to topologically sorts a graph specified by this *edge-list* representation. The function `(tsort edges)` should return a list that contains the labels in `edges`, in topological order.

    In topological sorting, the two symbols in an edge introduce a *precedence constraint*: for example, if `'(a b)` is an edge, then in the final sorted list of labels, `a` must precede `b`. As an example, `(tsort '((a b) (a c) (c b) (d b)))` can return either `'(a d c b)` or `'(a c d b)`.

    Not every graph can be topologically sorted; if the graph has a cycle, function `tsort` should call `error`.

    For details on topological sorting, see Sedgewick (1988, Chapter 32), or Knuth (1973, Section 2.2.3).

    ⟨*exercise transcripts* 154a⟩+≡

    ```
    -> (tsort '((duke commoner) (king duke) (queen duke) (country king) (god country)))
    (queen god country king duke commoner)
    ```

19. In the metacircular evaluator, the results of evaluating a top-level expression are not bound to `it`. Change the code in chunk 146d to correct this fault.

20. The real benefit of a metacircular evaluator is that it is easy to extend, so you can try out new language features. A particularly convenient method of extension is to use *syntactic sugar*, i.e., to define new constructs by rewriting them into existing constructs.

    (a) In full Scheme, `and` is *variadic*, and it works by *short-circuit evaluation.* This behavior can be expressed by the following laws:

    $$\texttt{(and)} \equiv \texttt{\#t}$$
    $$\texttt{(and } p_1\ p_2\ \ldots\ p_n\texttt{)} \equiv \texttt{(if } p_1\ \texttt{(and } p_2\ \ldots\ p_n\texttt{) \#f)}$$

    Use these laws and `foldr` to add `and` to the metacircular evaluator in Section 3.15.

    (b) Similarly, use `foldr` to add variadic, short-circuit `or` to the metacircular evaluator.

    (c) Add `let` to the metacircular evaluator using the law

    $$\texttt{(let ((}x_1\ e_1\texttt{) ... (}x_n\ e_n\texttt{)) e)} \equiv \texttt{((lambda (}x_1\ \ldots\ x_n\texttt{) e) }e_1\ \ldots\ e_n\texttt{)}$$

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

    (d) Add `let*` to the metacircular evaluator using the two laws

    $$\texttt{(let* () e)} \equiv \texttt{e}$$
    $$\texttt{(let* ((}x_1\ e_1\texttt{) ... (}x_n\ e_n\texttt{)) e)} \equiv \texttt{(let ((}x_1\ e_1\texttt{)) (let* (... (}x_n\ e_n\texttt{)) e))}$$

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

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

    $$\texttt{(letrec ((}x_1\ e_1\texttt{) ... (}x_n\ e_n\texttt{)) e)}$$

    to

    $$\texttt{(let ((}x_1\ \texttt{'()) ... (}x_n\ \texttt{'()))}$$
    $$\texttt{(begin (set }x_1\ e_1\texttt{) ... (set }x_n\ e_n\texttt{) e))}$$

    Use higher-order functions.

    (f) 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)`.

21. This exercise suggests some improvements to the metacircular evaluator in Section 3.15.

    (a) Instead of making `next` an ordinary key in the store, represent the store as a pair `(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 141b.)

    (b) Rewrite `bind` so that if a key does not appear in the a-list, it conses a new key-attribute pair onto the front of the a-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 21b, 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 problem, use the tests in chunks 147b and 147c.

22. Generalize the solver in Section 3.10.1 to handle *any* formula in the following form:

- Any symbol is a formula; the symbol stands for a variable.
- If f is a formula, the list `(not` f`)` is a formula.
- If $f_1, \ldots, f_n$ are formulas, the list `(and` f_1 ... f_n`)` is a formula.
- If $f_1, \ldots, f_n$ are formulas, the list `(or` f_1 ... f_n`)` is a formula.

Mathematically, the set of formulas F is the smallest set satisfying this equation:

$$\begin{aligned} F &= SYM \\ &\cup \{\texttt{(list2 'not } f\texttt{)} \mid f \in F\} \\ &\cup \{\texttt{(cons 'and } fs\texttt{)} \mid fs \in LIST(F)\} \\ &\cup \{\texttt{(cons 'or } fs\texttt{)} \mid fs \in LIST(F)\} \end{aligned}$$

Write a function `make-formula-true` that solves any formula in this form. Remember De Morgan's laws, one of which is mentioned on page 100.

The function `make-formula-true` should take three parameters: a formula, a failure continuation, and a success continuation. A call to `(make-formula-true f fail succ)` should attempt to find a satisfying assignment for formula `f`. If it finds a satisfying assignment, it should call `succ`, passing both the satisfying assignment (as an association list) and a resume continuation. If it fails to find a satisfying assignment, it should call `(fail)`.

You'll be able to use the ideas in Section 3.10.1, but probably not the code. Instead, try using `letrec` to define the following mutually recursive functions:

- `(make-formula formula bool cur fail succeed)` extends assignment `cur` to find an assignment that makes the single `formula` equal to `bool`.
- `(make-all formulas bool cur fail succeed)` extends `cur` to find an assignment that makes every formula in the list `formulas` equal to `bool`.
- `(make-any formulas bool cur fail succeed)` extends `cur` to find an assignment that makes any one of the `formulas` equal to `bool`.

In all the functions above, `bool` is `#t` or `#f`.

23. The discussion in section 3.16.5 only hints at the power of `call/cc`. Read and report on Friedman, Haynes, and Kohlbecker (1984), which contains many more examples.

24. The definition of `equal?` in chunk 69a doesn't work when passed function values. Considering the existing definition of equality and the following transcript, answer the two questions below.

⟨*transcript* 66⟩+≡
```
-> (= 3 3)
#t
-> (= 'a 'a)
#t
-> (= (cons 'a 'b) (cons 'a 'b))
#f
```

The two questions are these:

(a) What should `atom?` return when passed a procedure value?

(b) What should `equal?` do when passed a procedure value?

25. Steele and Sussman (1978) argue that, while static scope should certainly be the default scope rule, dynamically-scoped—or *fluid*—variables have their uses, as in this example: Suppose there is a function `print-num` to print integers, and you would like to make it possible to change the radix of printing. One way to accomplish this is to have `print-num` consult the global variable `RADIX`, so that a user who wants to print in base 8 does so in this way:

```
   ⋮
(set saveradix RADIX)
(set RADIX 8)
(print-num N)
(set RADIX saveradix)
   ⋮
```

In order not to surprise later clients of `print-num`, the code must save and restore `RADIX`. If `RADIX` were a dynamically-bound variable, the same effect could be obtained by defining:

```
(set print-8 (lambda (n) ( (lambda (RADIX) (print-num n)) 8)))
```

and replacing the code above by:

cons $\mathcal{P}$

```
   ⋮
(print-8 N)
   ⋮
```

Convinced by this and other examples of the usefulness of dynamically-scoped variables, some versions of Scheme include them in addition to ordinary, lexically-scoped ones. Explain this example and discuss whether you think the introduction of fluid variables into Scheme is justified. For more on fluid variables, see Abelson and Sussman (1985, pp. 321–325).

Learning about equational reasoning

The problems below are based on the proof techniques describe in Section 3.4.5 on page 77.

26. Prove that

```
(member x emptyset) = #f
```

27. Prove that for any predicate p?,

```
(exists? p? '()) = #f
```

28. Prove that for any list xs,

```
(all? (lambda (_) #t) xs) = #t
```

29. Prove that

```
(member x (add-element x s)) = #t
```

30. Prove that

```
(length (reverse xs)) = (length xs)
```

31. Prove that

```
(reverse (reverse xs)) = xs
```

32. Prove that

```
(append (append xs ys) zs) = (append xs (append ys zs))
```

33. Exercises 2c and 2d ask you to define functions `flatten` and `mirror`. Prove that

```
(flatten (mirror xs)) = (reverse (flatten xs))
```

34. Prove that the composition of maps is the map of the composition:

```
(map f (map g xs)) = (map (o f g) xs)
```

35. Prove again that the composition of maps is the map of the composition, but without an explicit list arguments.

```
(o ((curry map) f) ((curry map) g)) = ((curry map) (o f g))
```

    To prove that two functions are equal, show that when applied to equal arguments, they always return equal results.

36. Exercise 5 defines functions `takewhile` and `dropwhile`. Prove that for any list xs,

```
(append (takewhile p? xs) (dropwhile p? xs)) = xs
```

Learning about the operational semantics

37. In both Scheme and μScheme, a `val` binding of a name that is already bound is equivalent to `set`. It would be much easier if `val` always created a new binding.

    (a) Write a DEFINEGLOBAL rule to show the operational semantics of such a `val`.

    (b) Write a μScheme program to detect whether `val` uses the Scheme semantics or the new semantics, and explain how it works.

    (c) Compare and contrast the two ways of defining `val`. Which design do you prefer, and why?

38. The behavior of the DEFINENEWGLOBAL rule may strike you as rather odd, as it permits a "definition" like

    ⟨*transcript* 66⟩+≡
    ```
    -> (val u u)
    ```

    for a previously undefined `u`. The definition is valid, and `u` has a value, although the value is not specified. Similar behavior is typical of a number of dynamically typed languages, such as Awk, Icon, and Perl, in which a new variable—with a well-specified value, even—can be called into existence just by referring to it.

    In μScheme, this rule makes it easy to define recursive functions, as explained on page 122. But you might prefer a semantics in which whenever $\mathtt{u} \notin \mathrm{dom}\,\rho$, `(val u u)` is rejected.

    ⟨*imaginary transcript* 166b⟩≡
    ```
    -> (val u u)
    error: variable u not found
    ```

    This exercise is to rewrite the semantics of μScheme so that `(val u u)` is rejected but it is still possible to define recursive functions.

39. The metacircular `val` in Section 3.15.5 carefully distinguishes an undefined global variable from a global variable that is bound to *nil*. Is it possible to write a short μScheme program that gives a different answer if we treat global variables bound to *nil* as if they were undefined? If so, write such a program. If not, explain why not.

    Hint: mutation.

40. Although `letrec` is convenient, it is not required. Here again is the rule for evaluating `letrec`:

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma \text{ (and all distinct)} \\ x_1, \ldots, x_n \text{ all distinct} \\ \rho' = \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\} \qquad \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \ldots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(LETREC)}$$

In this exercise, you show that `letrec` is not necessary.

(a) Write a translation from an expression that uses `letrec` to one that does not use `letrec`. Other forms of expression that you might find especially useful include `let`, `begin`, and `set`.

(b) Using the operational semantics, prove that your implementation of `letrec` is equivalent to the original, at least when the names defined in the `letrec` don't depend on unspecified values.

(c) Argue whether you think `letrec` should be removed from μScheme or kept, and why.

41. Write rules of operational semantics for full Scheme primitives `set-car!` and `set-cdr!`.

42. Ben User decides to simplify the operational semantics by writing rules this way:

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \text{PRIMITIVE}(\texttt{cons}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle v_1, \sigma_2\rangle \\ \langle e_2, \rho, \sigma_2\rangle \Downarrow \langle v_2, \sigma_3\rangle\end{array}}{\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma_0\rangle \Downarrow \langle \text{CONS}\langle v_1, v_2\rangle, \sigma_3\rangle} \quad \text{(Ben User's CONS)}$$

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \text{PRIMITIVE}(\texttt{car}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle \text{CONS}\langle v_1, v_2\rangle, \sigma_2\rangle\end{array}}{\langle \text{APPLY}(e, e_1), \rho, \sigma_0\rangle \Downarrow \langle v_1, \sigma_3\rangle} \quad \text{(Ben User's CAR)}$$

$$\frac{\begin{array}{c}\langle e, \rho, \sigma_0\rangle \Downarrow \langle \text{PRIMITIVE}(\texttt{cdr}), \sigma_1\rangle \\ \langle e_1, \rho, \sigma_1\rangle \Downarrow \langle \text{CONS}\langle v_1, v_2\rangle, \sigma_2\rangle\end{array}}{\langle \text{APPLY}(e, e_1), \rho, \sigma_0\rangle \Downarrow \langle v_2, \sigma_3\rangle} \quad \text{(Ben User's CDR)}$$

Under these rules, what primitives of full Scheme become very difficult to specify?

43. Use the operational semantics in this chapter and in Chapter 2 to prove that if a function in Impcore produces an answer, then if that same function is passed the same values in μScheme, it produces the same answer. Assume that every name used in the function is either a formal parameter or the name of a function, but not both.

You will need two inductions: the first induction is on the number of top-level functions known to the interpreters. For the basis case of this induction, you may assume that all the primitives satisfy the theorem. For the induction step, you will need to show that if all existing functions satisfy the theorem, so does a new user-defined function.

The second induction will help you prove the induction step of the first induction: that a new function satisfies the theorem. Here, it will be easiest to prove an inductive lemma on the function body, which says that if two bodies are evaluated in corresponding environments, they produce corresponding answers. For this lemma, you will want to use structural induction on the abstract-syntax tree of the body.

44. Use the operational semantics to prove that variables in μScheme cannot alias. That is, prove that during the evaluation of a μScheme program, we never construct an environment ρ such that $x \neq y$ but $\rho(x) = \rho(y)$.

    Hint: it will help to prove that any location in the range of ρ is also in the domain of σ.

Learning about the interpreter

45. This exercise examines the implementation of "unspecified" values using the function `unspecified`. In particular, I ask you to investigate another way of preventing programmers from writing code that depends on unspecified values.

    (a) Prove, by examining the code, that the only way a computation can depend on the result of calling `unspecified()` is by fetching the unspecified value from a location that is bound in an environment.

    (b) Extend the definition of `Value` in chunk 114c to include an `UNSPECIFIED` alternative. (Or slip the extra alternative directly into type `Valuealt` in the `all.h` file for the interpreter.) Then, modify function `unspecified` so it simply returns a value tagged `UNSPECIFIED`.

    (c) Change the code that evaluates `VAR` so that if a variable is bound to a location that contains `UNSPECIFIED`, `eval` calls `error`.

    (d) Prove that using the modified interpreter, any computation that depends on an unspecified value halts with an error message.

    (e) Prove that using the modified interpreter, any computation that *does not* depend on an unspecified value behaves the same way as in the original interpreter.

    (f) Evaluate both the performance of the modified interpreter and the effort it took you to change the code. Is the cost of checking every variable lookup outweighed by the savings in `val` and `letrec`? Is the ability to detect ill-behaved computations worth the extra programming effort?

    (g) Why do you think full Scheme includes something like "unspecified" in its semantics? What do you suppose an industrial Scheme compiler like Chez Scheme does with "unspecified" values?

46. This problem is about the representation of environments in Section B.2.1. Call "environment depth" the length of a chain of `tl` pointers. Argue that there is a fixed maximum environment depth of any environment created during a computation, and that it can be determined from the text of the Scheme program. (Instrumenting the interpreter to compute this value may help you answer this question.)

47. Add the new primitive `list`, which should accept any number of arguments.

48. Add primitives `set-car!` and `set-cdr!`. Remember that `set-car!` does *not* change the value the `car` field of a cons cell; it replaces the contents of the location that the `car` field points to. You'll have it right if your mutations are visible through different variables:

⟨*mutation transcript* 169⟩≡
```
-> (val q '())
-> (val p '(a b c))
-> (set q p)
-> (set-car! p 'x)
-> (car q)
x
```

Once you have defined `set-car!` and `set-cdr!`, exercise them as follows:

(a) Define `rplac-bind`, which replaces a binding in place instead of creating a new one.

(b) Redefine `store` in the metacircular evaluator (Section 3.15) to use `rplac-bind`, and measure the resulting speedup.

(c) Use `set-cdr!` to write `nreverse`, which destructively reverses a list *in place*, i.e., with no `cons`es.

49. Add a `read` primitive (as in Exercise 22 on page 57 of Chapter 2). Inside your `read` primitive, you may find it helpful to call `p = readpar(...)`, followed by `mkPL(mkAtom(strtoname("quote")), mkPL(p, NULL))`.

Use the `read` primitive to build an interactive version of the metacircular interpreter.

50. Add a trace facility to the interpreter. Whenever a traced function is called, its name (if any) and arguments should be printed to the terminal. (If the name of a function is not known, print a representation of its abstract syntax.) When a traced function returns, the function and its result should be printed. The trace output should be clearly labeled and should use indentation to indicate calls and returns.

Choose one of the following two methods to indicate which functions to trace:

(a) Provide primitives to turn tracing of individual functions on and off.

(b) Use variable `&trace` as a "trace count." (Simply look it up in the current environment.) While `&trace` is bound to a location containing a nonzero number, each call and return should decrement the trace count and print a line. If the trace count is negative, tracing runs indefinitely.

Trace `length` (as we did on page 68), then `sieve` and `remove-multiples`.

Printing the abstract syntax for a function provides good intuition, but it may be more helpful to print non-global functions in closure form.

(c) When calling a closure, print it in closure form instead of printing its name. Which of the two methods is better?

51. If errors occur during `use`, our interpreter can leak open file descriptors.

(a) Devise an `onerror` interface that registers an action to perform when an error occurs. That action should be represented by a closure, i.e., a pair consisting of a function plus a pointer to potential free variables. The μScheme representation of closures is probably not appropriate; devise one that is convenient for C programmers.

(b) Devise an extension that enables you to rescind a registered action.

(c) Use your new interface to ensure that the μScheme interpreter does not leak open file descriptors.

(d) If you have done the trace exercise (Exercise 50), use it to reset the indentation to column 0 on error.

52. The variable-handling code in chunks 129c and 129d calls `find` twice for both `var` and `set`. This code offends some readers.

(a) Change the code in chunks 129c and 129d so that each chunk calls `find` exactly once.

(b) Change the implementation of `find` (in chunk 604b) to have a one-element cache, so that if `find` is called twice in a row with the same arguments, the second call returns in constant time.

(c) Measure the speedups obtained by each of these improvements separately and by the combination of the two improvements. What can you conclude from your measurements?

53. Our interpreter supports standard Scheme dot notation for output, but not for input. Modify the interpreter so it supports dot notation for input—if the `cdr` of a cons cell is not a cons cell or *nil*, then the `car` and `cdr` are separated by a dot. The interpreter already uses this notation for output. On output, a dot can appear only before the *last number or symbol* in a list; this rule applies to sublists as well. Here are some examples:

cons $\mathcal{P}$

⟨*transcript* 66⟩+≡

```
-> (cons 'a (cons 'b (cons 4 (cons 5 '()))))
(a b 4 5)
-> (cons 'a (cons 'b (cons 4 5)))
(a b 4 . 5)
-> (cons (cons 'a 'b) (cons 4 5))
((a . b) 4 . 5)
```

This rule minimizes the number of dots used by output.

For input, a dot can appear only before the *last item* in a list (though this last item can be a list), so `'(a b . c d)` is illegal. The dot represents a cons cell pointing to the items before and after the dot. Thus, `'(a b c . d)` represents the list `(cons 'a (cons 'b (cons 'c 'd)))`, and `'(a b . (c d))` the list `(cons 'a (cons 'b '(c d)))`, which is the same as `'(a b c d)`.