Appendix B
FUNCTIONAL PROGRAMMING WITH SCHEME

The languages usually studied in computer science—namely, Pascal, C, Modula-2, and Ada—are considered imperative languages because the basic construct is a command. These languages are heavily influenced by the “von Neumann architecture” of computers, which includes a store (memory) and an instruction counter used to identify the next instruction to be fetched from the store. The computation model has control structures that determine the sequencing of instructions, which use assignments to make incremental modifications to the store.

Imperative languages are characterized by the following properties:

- The principal operation is the assignment of values to variables.
- Programs are command oriented, and they carry out algorithms with statement level sequence control, usually by selection and repetition.
- Programs are organized as blocks, and data control is dominated by scope rules.
- Computing is done by effect, namely by changes to the store.

The computing by effect intrinsic to imperative programming plays havoc with some of the mathematical properties that are essential to proving the correctness of programs. For example, is addition commutative in an imperative program? Does “write(a+b)” always produce the same value as “write(b+a)”? Consider the following Pascal program:

```
program P (output);
    var b : integer;
    function a : integer;
            begin b := b+2; a := 5 end;
    begin
            b := 10
            write(a+b)  or   write(b+a)
    end.
```

In fact, implementations of Pascal will most likely give different results for the two versions of this program, depending on the order of evaluation of expressions. This anomaly is caused by the side effect in the expression being evaluated, but programming by effect lies at the heart of imperative programming. If we depend on imperative programs, we must discard many of the basic properties of mathematics, such as associative and commuative laws of addition and multiplication and the distributive law for multiplication over addition.

The functional programming paradigm provides an alternative notion of programming that avoids the problems of side effects. Functional languages are concerned with data objects and values instead of variables. Values are bound to identifiers, but once made, these bindings cannot change. The principal operation is function application. Functions are treated as first-class objects that may be stored in data structures, passed as parameters, and returned as function results. A functional language supplies primitive functions, and the programmer uses function constructors to define new functions. Program execution consists of the evaluation of an expression, and sequence control depends primarily on selection and recursion. A pure functional language has no assignment command; values are communicated by the use of parameters to functions. These restrictions enforce a discipline on the programmer that avoids side effects. We say that functional languages are referentially transparent.

Principle of Referential Transparency: The value of a function is determined by the values of its arguments and the context in which the function application appears, and it is independent of the history of the execution. ∎

Since the evaluation of a function with the same argument produces the same value every time that it is invoked, an expression will produce the same value each time it is evaluated in a given context. Referential transparency guarantees the validity of the property of substituting equals for equals.

Lisp

Work on Lisp (**Lis**t **p**rocessing) started in 1956 with an artificial intelligence group at MIT under John McCarthy. The language was implemented by McCarthy and his students in 1960 on an IBM 704, which also had the first Fortran implementation. Lisp was an early example of interactive computing, which played a substantial role in its popularity. The original development of Lisp used S-expressions (S standing for symbolic language) with the intention of developing an Algol-like version (Lisp 2) with M-expressions (M for metalanguage). When a Lisp interpreter was written in Lisp with S-expressions, Lisp 2 was dropped. The principal versions, which are based on Lisp 1.5, include Interlisp, Franz Lisp, MacLisp, Common Lisp, and Scheme.

Lisp has a high-level notation for lists. Functions are defined as expressions, and repetitive tasks are performed mostly by recursion. Parameters are passed to functions by value. A Lisp program consists of a set of function definitions followed by a list of expressions that may include function evaluations.

Scheme Syntax

The Scheme version of Lisp has been chosen here because of its small size and its uniform treatment of functions. In this appendix we introduce the fundamentals of functional programming in Scheme. When we say Scheme, we are referring to Lisp. The basic objects in Scheme, called S-expressions, consist of atoms and "dotted pairs":

<S-expr> ::= <atom> | **(** <S-expr> **.** <S-expr> **)**

The only terminal symbols in these productions are the parentheses and the dot (period). The important characteristic of an S-expression is that it is an atom or a pair of S-expressions. The syntactic representation of a pair is not crucial to the basic notion of constructing pairs.

Atoms serve as the elementary objects in Scheme. They are considered indivisible with no internal structure.

<atom> ::= <literal atom> | <numeric atom>

<literal atom> ::= <letter> | <literal atom> <letter> | <literal atom> <digit>

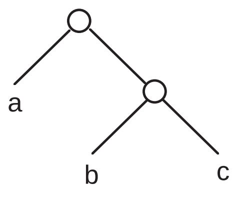
<numeric atom> ::= <numeral> | – <numeral>

<numeral> ::= <digit> | <numeral> <digit>

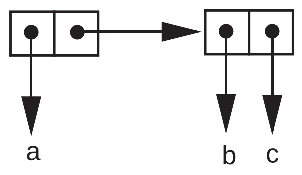
Literal atoms consist of a string of alphanumeric characters usually starting with a letter. Most Lisp systems allow any special characters in literal atoms as long as they cannot be confused with numbers. The numeric atoms defined here represent only integers, but most Lisp systems allow floating-point numeric atoms.

Since S-expressions can have arbitrary nesting when pairs are constructed, Scheme programmers rely on a graphical representation of S-expressions to display their structure. Consider the following diagrams illustrating the S-expression (a . (b. c)):

Lisp tree (or L-tree):



Cell diagram (or box notation):



We prefer using the box notation for S-expressions. Atoms are represented as themselves, and if the same atom is used twice in an S-expression, a single value can be shared since atoms have unique occurrences in S-expressions.

Functions on S-expressions

The simplicity of Scheme (and Lisp) derives from its dependence on several basic functions for constructing pairs and selecting components of a pair. Two selector functions are used to investigate a pair:

car	Applied to a nonatomic S-expression, car returns the left part.
cdr	Applied to a nonatomic S-expression, cdr returns the right part.

On the IBM 704, car stood for "**c**ontents of **a**ddress **r**egister" and cdr for "**c**ontents of **d**ecrement **r**egister". Some authors have suggested that "head" and "tail" or "first" and "rest" are more suggestive names for these functions, but most Lisp programmers still use the traditional names.

The following examples that use brackets [ ] to delimit arguments do not follow correct Scheme syntax, which will be introduced shortly:

car [((a . b) . c)] = (a . b)

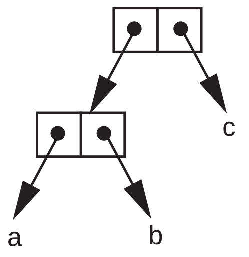
cdr [((a . b) . c)] = c

An error results if either function is applied to an atom.

An abstract implementation of the selector functions can be explained in terms of a box diagram:

car returns the left pointer.

cdr returns the right pointer.



A single constructor function cons builds a pair given two S-expressions:

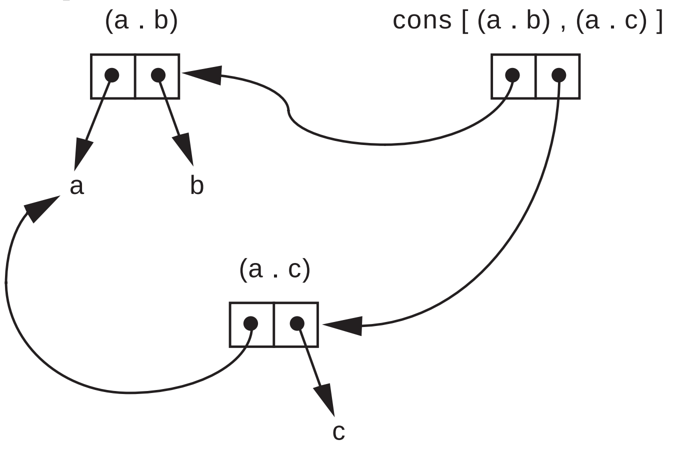
cons	Applied to two S-expressions, cons returns a dotted pair containing them.

For example:

cons[p , q] = (p . q)

cons[(a . b) , (a . c)] = ((a . b) . (a . c))

As an abstract implementation, we allocate a new cell and set its left and right components to point to the two arguments. Observe that the atom a is shared by the two pairs.



Lists in Scheme

The notion of an S-expression is too general for most computing tasks, so Scheme primarily deals with a subset of the S-expressions. A list in Scheme is an S-expression with one of two forms:

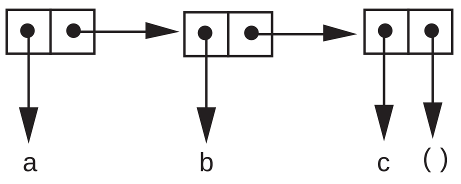
1. The special atom ( ) is a list representing the empty list. Note that ( ) is the only S-expression that is both an atom and a list.
2. A dotted pair is a list if its right (cdr) element is a list.

S-expressions that are lists use special notation:

(a . ( ))	is represented by	(a)
(b . (a . ( )))	is represented by	(b a)
(c . (b . (a . ( ))))	is represented by	(c b a)

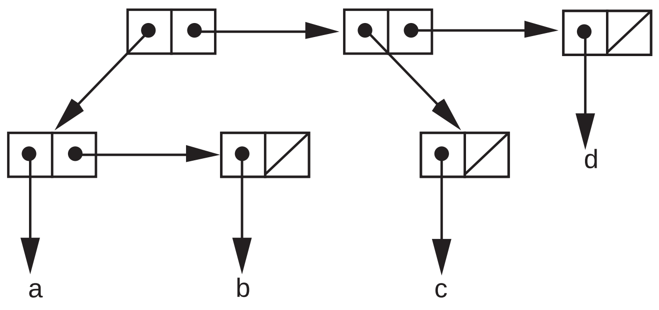
Cell diagrams for lists are usually drawn with a horizontal “spine” that stretches from left to right. The spine contains as many boxes as the list has elements at its top level.

(a b c)



((a b) (c) d) =

((a . (b . ( ))) . ((c . ( )) . (d . ( ))))



Observe the abbreviation of a slash through the cell at the end of a list to represent a pointer to an empty list ( ).

The elementary constructor and selectors for S-expressions have special properties when applied to lists.

car	When applied to a nonempty list, car returns the first element of the list.
cdr	When applied to a nonempty list, cdr returns a copy of the list with the first element removed.
cons	When applied to an arbitrary S-expression and a list, cons returns the list obtained by appending the first argument onto the beginning of the list (the second argument).

For example:

```
car [ (a b c) ] = a                 cdr [ (a b c) ] = (b c)
car [ ((a)) ] = (a)                 cdr [ ((a)) ] = ( )
cons [(a) , (b c) ] = ((a) b c)     cons [ a , ( ) ] = (a)
```

Syntax for Functions

In Scheme, the application of a function to a set of arguments is expressed as a list:

(function-name sequence-of-arguments)

This prefix notation is known as **Cambridge Polish Form** since it was developed at MIT in Cambridge. We illustrate the notation by introducing many of the predefined Scheme numeric functions.

Unary functions:

(add1 5)	returns	6
(sub1 0)	returns	-1
(abs (add1 -5))	returns	4

Binary functions:

(- 6 9)	returns	-3
(quotient 17 5)	returns	3
(/ 17 5)	returns	3.4
(* 10 12)	returns	120
(- (* 10 2) (+ 13 3))	returns	4
(modulo 53 5)	returns	3

N-ary functions:

(+ 1 2 3 4 5)	returns	15
(* 1 2 3 4 5)	returns	120
(max 2 12 3 10)	returns	12
(min (* 4 6) (+ 4 6) (- 4 6))	returns	-2

Miscellaneous functions:

(expt 2 5)	returns	32
(expt 5 -2)	returns	0.04
(sqrt 25)	returns	5
(sqrt 2)	returns	1.4142135623730951

Functions that return Boolean values are called predicates. In Scheme predicates return either the atom #t, which stands for true, or #f, the value for false. Scheme programmers usually follow the convention of naming a predicate with identifiers that end in a question mark.

(negative? -6)	returns	#t	(= 6 2)	returns	#f
(zero? 44)	returns	#f	(< 0.5 0.1)	returns	#f
(positive? -33)	returns	#f	(>= 3 30)	returns	#f
(number? 5)	returns	#t	(<= -5 -3)	returns	#t
(integer? 3.7)	returns	#f	(odd? 5)	returns	#t
(real? 82)	returns	#f	(even? 37)	returns	#f
(> 6 2)	returns	#t			

Scheme Evaluation

When the Scheme interpreter encounters an atom, it evaluates the atom:

- Numeric atoms evaluate to themselves.

- The literal atoms #t and #f evaluate to themselves.
- Each other literal atom evaluates to the value, if any, that has been bound to it.

The define operation may be used to bind a value to an atom. The operation makes the binding and returns a value:

(define a 5)	returns a
(define b 3)	returns b
a	returns 5
(+ a b)	returns 8
(+ a c)	returns an error since c has no value bound to it.

Although the value returned by define is unspecified in the Scheme standard, most Schemes return the name of the identifier that has just been bound.

When the Scheme interpreter evaluates a list, it expects the first item in the list to be an expression that represents a function. The rest of the items in the list are evaluated and given to the function as argument values.

(* a (add1 b)) returns 20

Suppose now that we want to apply car to the list (a b c). Evaluating the expression (car (a b c)) means that a must represent a function, which will be applied to the values of b and c, and the resulting value is passed to car. Since we want to apply car to the list (a b c) without evaluating the list, we need a way to suppress that evaluation. Scheme evaluation is inhibited by the quote operation.

(quote a)	returns the symbol a
(quote (a b c))	returns (a b c) unevaluated
(car (quote (a b c)))	returns a
(cdr (quote (a b c)))	returns (b c)
(cons (quote x) (quote (y z)))	returns the list (x y z)

The quote operation may be abbreviated by using an apostrophe.

(cdr '((a) (b) (c)))	returns ((b) (c))
(cons 'p '(q))	returns (p q)
(number? 'a)	returns #f
'a	returns a
'(1 2 3)	returns (1 2 3)

The car and cdr functions may be abbreviated to simplify expressions. (car (cdr '(a b c))) may be abbreviated as (cadr '(a b c)). Any combination of a's and d's between c and r (up to four operations) defines a Scheme selector function.

Now that we have a mechanism for suppressing evaluation of a literal atom or a list, several more fundamental functions can be described.

pair?	When applied to any S-expression, pair? returns #t if it is a dotted pair, #f otherwise.

```
(pair? 'x)    returns  #f
(pair? '(x))  returns  #t
```

atom?	When applied to any S-expression, atom? is the logical negation of pair?. (atom? is not standard in Scheme.)
null?	When applied to any S-expression, null? returns #t if it is the empty list, #f otherwise.

```
(null? '( ))    returns  #t
(null? '(( )))  returns  #f
```

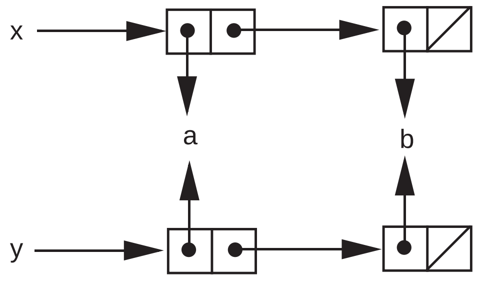
eq?	When applied to two *literal atoms*, eq? returns #t if they are the same, #f otherwise.

```
(eq? 'xy 'x)              returns  #f
(eq? (pair? 'gonzo) #f)   returns  #t
(eq? '(foo) '(foo))       returns  #f
```

The reader may find the equality function eq? somewhat confusing since it may appear that the expression (foo) should be equal to itself. To explain this unusual version of equality, we develop a short example. We use the define operation to create two bindings.

```
(define x '(a b))
(define y '(a b))
```

To explain why (eq? x y) returns #f, consider the cell diagram below. Each time the Scheme interpreter processes an S-expression, such as (define x '(a b)), it creates a new copy of the structure being processed.



Although the values appear to be the same, they are two different copies of the same S-expression. The test (eq? x y) returns #f because x and y point to two different objects. We can view eq? as testing pointer equality. On atoms eq? acts as an equality test since atoms are treated as unique objects. The equality of numeric atoms can be tested using the = function. The equality of general S-expressions will be considered later.

Special Forms

All the operations considered so far do not act in the same way. Scheme functions, such as +, car, null?, =, and user-defined functions, always evaluate their arguments. In fact, when (+ (car '(2 4 6)) 5) is submitted to the interpreter, each of the expressions +, (car '(2 4 6)), and 5 are evaluated:

+	evaluates to the predefined addition operation,
(car '(2 4 6))	evaluates to the number 2, and
5	evaluates to the number 5.

On the other hand, several of the operations described so far do not and cannot evaluate all of their operands. (quote a) simply returns its operand unevaluated. (define x (+ 5 6)) evaluates its second argument, but leaves its first argument unevaluated.

These operations are called **special forms** to distinguish them from normal Scheme functions. A complete list of the special forms in Scheme follows:

and	delay	let	quasiquote
begin	do	let*	quote
case	if	letrec	set!
cond	lambda	or	while
define			

For some of these special forms, the determination of which arguments are evaluated is made on a dynamic basis using the results of evaluations performed so far. We will not take the time to describe all of the special forms in Scheme. The description of those not used in this appendix can be found in the references for Scheme.

Defining Functions in Scheme

The special form define returns the name of the function (or other object) being defined; more importantly, it has the side effect of binding an object that may be a function to the name.

```
(define name (lambda (list-of-parameters) expression))
```

The use of lambda here will be explained later. The basic idea is that executing the function defined by the expression (lambda (list-of-parameters) expression) involves evaluating the expression in an environment that contains binding of the parameters in the list to actual arguments. We give examples to illustrate user-defined Scheme functions below.

- Calculate the hypotenuse given the legs of a right triangle.

```
        (define hypotenuse (lambda (a b) (sqrt (+ (* a a) (* b b))) ))
    (hypotenuse 3 4)        returns 5.0
    (hypotenuse 10 20)      returns 22.360679774997898
```

- Find the first item in a list (a synonym for car).

```
        (define  first  (lambda (L)  (car L)))
    (first '((a b c)))  returns  (a b c)
```

- Find the second item in a list.

```
        (define  second  (lambda (L)  (cadr L) ))
    (second '((a) (b) (c)))  returns  (b)
```

What if the value bound to L does not have a first or second element? We use revisions to these two functions to illustrate conditional expressions in Scheme. We plan to change the definition so that

If L is empty, both functions return #f.

If L has only one element, second returns #f.

A mechanism for making decisions is needed to carry out these revisions. Decisions in Scheme are represented as conditional expressions using the special form cond:

(cond (c_1 e_1) (c_2 e_2) … (c_n e_n) (else e_{n+1})),

which is equivalent to

```
if c1 then return e1
else if c2 then return e2
        :
else if cn then return en
else return en+1
```

If all of c_1, c_2, …, c_n are false and the else clause is omitted, the cond result is unspecified, although many implementations return an empty list. The function cond is a special form since it does not evaluate all its arguments. For the purposes of testing, any non-#f value represents true.

Now we use cond to revise the definitions of the functions first and second.

```
(define  first  (lambda (L)
    (cond  ((null? L)  #f)
           (else  (cdr L))  )))
```

```
(define  second  (lambda (L)
      (cond  ((null? L)  #f)
             ((null? (cdr L))  #f)
             (else  (cadr L))  )))
```

Both cond and the body of function definitions allow more generality, allowing a sequence of expressions. Each expression is evaluated and the value of the last one is the result returned. The other expressions are evaluated for their side effects (a non-functional aspect of Scheme).

```
(define  categorize  (lambda (n)
      (cond  ((= n 0)  (display 'zero) 0)
             ((positive? n)  (display 'positive) 1)
             (else  (display 'negative) -1)) ))
```

Another special form for decision making is the if operation:

```
(if test then-expression else-expression)
```

For example,

```
(define  safe-divide  (lambda (m n)
     (if  (zero? n)
          0
          (/ m n)) ))
```

Recursive Definitions

The main control structure in Scheme is recursion. Functions that require performing some sequence of operations an arbitrary number of times can be defined inductively. These definitions translate directly into recursive definitons in Scheme. In the next two examples, we define a function using mathematical induction and then translate that definition using recursion.

- Exponentiation (assume $m \neq 0$)

$$m^0 = 1$$
$$m^n = m \bullet m^{n-1} \text{ for } n>0$$

```
(define  power  (lambda (m n)
                   (if  (zero? n)
                        1
                        (*  m (power m (sub1 n))) )))
```

A sample execution of the power function demonstrates how the recursion unfolds. In reality, the induction hypothesis inherent in a recursion definition ensures that the result computes what we want.

```
(power  2  3)
         =  2 • (power  2  2)
                  =  2 • [2 • (power  2  1)]
```

```
                        =  2 • [2 • [2 • (power  2  0)]]
                                =  2 • [2 • [2 • 1]]
                                        = 2 • [2 • 2]   =   2 • 4   =   8
```

• Fibonacci

fib(0) = 1

fib(1) = 1

fib(n) = fib(n-1) + fib(n-2) for n>1

```
(define  fib  (lambda (n)
              (cond    ((zero? n)  1)
                       ((zero? (sub1 n))  1)
                       (else  (+  (fib (sub1 n))  (fib (- n 2)) )) )))
```

Lambda Notation

Scheme contains a mechanism for defining anonymous functions, as was the case in the lambda calculus (see Chapter 5). The lambda expression $\lambda x,y \,.\, y^2+x$ becomes the S-expression (lambda (x y) (+ (* y y) x)) in Scheme. An anonymous function can appear anywhere that a function identifier is allowed. For example, we can apply the previous function as follows:

((lambda (x y) (+ (* y y) x)) 3 4) returns 19.

In fact, the expression that we use to define a function is simply making a binding of an identifier to a lambda expression representing an anonymous function. For example, the expression (define fun (lambda (x y) (+ (* y y) x))) binds the name fun to the anonymous function (lambda (x y) (+ (* y y) x))). Scheme permits an abbreviation of such a definition using notation that shows the pattern of a call of the function as in

(define (fun x y) (+ (* y y) x)).

Recursive Functions on Lists

Many functions in Scheme manipulate lists. Therefore we develop three examples that show the basic techniques of processing a list recursively.

1. Count the number of occurrences of atoms in a list of atoms. For example, (count1 '(a b c b a)) returns 5.

   Case 1 : List is empty ⇒ return 0

   Case 2 : List is not empty

   ⇒ it has a first element that is an atom

   ⇒ return (1 + number of atoms in the cdr of the list).

In Scheme, cond can be used to select one of the two cases.

```
(define  count1  (lambda (L)
                        (cond  ((null? L)  0)
                               (else  (add1 (count1 (cdr L)))) )))
```

2. Count the number of occurrences of atoms at the “top level” in an arbitrary list. For example, (count2 '(a (b c) d a)) returns 3.

   Case 1 : List is empty ⇒ return 0

   Case 2 : List is not empty.

   Subcase a : First element is an atom (it not is a pair)
   ⇒ return (1 + number of atoms in the cdr of the list).

   Subcase b : First element is not an atom
   ⇒ return the number of atoms in the cdr of the list.

   We write this algorithm in Scheme as the function

```
(define  count2  (lambda (L)
                        (cond  ((null? L)  0)
                               ((atom? (car L))  (add1 (count2 (cdr L))))
                               (else  (count2 (cdr L))) )))
```

3. Count the number of occurrences of atoms at all levels in an arbitrary list. For example, (count2 '(a (b c) d (a))) returns 5.

   Case 1 : List is empty ⇒ return 0

   Case 2 : List is not empty.

   Subcase a : First element is an atom
   ⇒ return (1 + number of atoms in the cdr of the list).

   Subcase b : First element is not an atom
   ⇒ return (the number of atoms in the car of the list
   + the number of atoms in the cdr of the list).

   The corresponding Scheme function is defined below.

```
(define  count3  (lambda (L)
                        (cond  ((null? L)  0)
                               ((atom? (car L))  (add1 (count3 (cdr L))))
                               (else  (+ (count3 (car L)) (count3 (cdr L)) )) )))
```

Now that we have seen the basic patterns for defining functions that process lists, we describe a number of useful list manipulation functions, most of which are predefined in Scheme. We give them as user-defined functions as a means of explaining their semantics and to provide additional examples of Scheme code. In many Scheme systems the identifiers associated with predefined functions may not be redefined since they are reserved words. Therefore the names of the following user-defined functions may have to be altered to avoid confusion.

- **Length of a list**

```
(define  length  (lambda (L)
                    (if  (null? L)
                         0
                         (add1 (length (cdr L)))  )))
```

The function length will work identically to the predefined length function in Scheme except that the execution may be slower or a stack may overflow for long lists since the predefined functions may be more efficiently implemented.

- **The nth element of a list**

```
(define  nth  (lambda (n L)
                    (if   (zero? n)
                          (car L)
                          (nth  (sub1 n) (cdr L)) )))
```

This function finds the n^th^ element of a list using zero as the position of the first item. So the first element is called the 0th.

- **Equality of arbitrary S-expr essions**

The strategy for the equality function is to use = for numeric atoms, eq? for literal atoms, and recursion to compare the left parts and right parts of dotted pairs. The corresponding predefined function is called equal?.

```
(define  equal? (lambda (s1 s2)
                    (cond  ((number? s1)  (= s1 s2))
                           ((atom? s1)  (eq? s1 s2))
                           ((atom? s2)  #f)
                           ((equal?(car s1) (car s2))  (equal? (cdr s1) (cdr s2)))
                           (else  #f)  )))
```

- **Concatenate two lists**

```
(define  concat  (lambda (L1 L2)
                    (cond  ((null? L1)  L2)
                           (else  (cons (car L1) (concat (cdr L1) L2))) )))
```

For example, (concat '(a b c) '(d e)) becomes

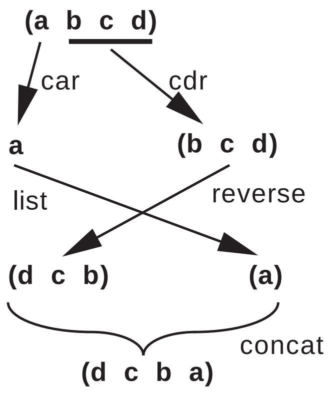
```
        (cons 'a (concat '(b c) '(d e)))
                   = (cons 'a (cons 'b (concat '(c) '(d e))))
                             = (cons 'a (cons 'b (cons 'c (concat '( ) '(d e)))))
                                       = (cons 'a (cons 'b (cons 'c '(d e))))
                                                 =  (cons 'a (cons 'b '(c d e)))
                                                           =  (cons 'a '(b c d e))
                                                                     =  (a b c d e)
```

Although its name may suggest otherwise, this is a pure function, so neither argument is altered. If length(L1) = n, concat requires n applications of cons; this is a measure of how much work is done. The predefined function for concatenating lists is called append and allows an arbitrary number of lists as its arguments. User functions with an arbitrary number of arguments can be defined several ways, but that topic is beyond the scope of this presentation.

- **Reverse a list**

```
(define  reverse  (lambda (L)
                       (if   (null? L)
                             '( )
                             (concat (reverse (cdr L)) (list (car L))) )))
```

The diagram below shows the way reverse handles a list with four elements. Observe that we assume that the function works correctly on lists of length three (the induction hypothesis).



- **Membership (at the top level) in a list**

```
(define  member  (lambda (e L)
                       (cond  ((null? L)  #f)
                              ((equal? e (car L))  L)
                              (else  (member e (cdr L))) )))
```

We might expect this Boolean function to return #t (true) or #f (false), but it returns the rest of the list starting with the matched element for true. This behavior is consistent with the interpretation that any non-#f object represents true. If the item is not in the list (the first case in the cond expression), member returns #f.

- **Logical operations**

```
(define and (lambda (s1 s2) (if s1 s2 #f) ))
(define or (lambda (s1 s2) (if s1 s2 #t) ))
```

The predefined "and" and "or" operations (actually special forms) allow an arbitrary number of S-expressions as arguments. In these functions and in our user-defined functions, the arguments are tested from left to right until a decision can be made. For and, the first false argument makes the result #f. For or, the first true argument makes the result non-#f.

Since we defined and and or as regular functions, all of the arguments in a call must be evaluated even if they are not all needed. The special forms and and or evaluate only as many operands as are needed to make a decision.

Scope Rules in Scheme

In Lisp 1.5 and many of its successors, access to nonlocal identifiers is resolved by dynamic scoping: the calling chain (along dynamic links) is followed until the identifier is found local to a program unit (a function in Lisp). McCarthy claims that he intended for Lisp to have static scoping but that a mistake was made in implementing the early versions of Lisp(see [Wexelblat81]). In fact, dynamic scoping is easier to implement for Lisp.

Scheme and Common Lisp use static scoping; nonlocal references in a function are resolved at the point of function definition. Static scoping is implemented by associating a closure (instruction pointer and environment pointer) with each function as it is defined. The calling stack maintains static links for nonlocal references.

Top-level define's create a global environment composed of the identifiers being defined that is visible everywhere. A new scope is created in Scheme when the formal parameters, which are local identifiers, are bound to actual values when a function is invoked. The following transcript shows the creation of a global identifier a and a local (to f) identifier a.

```
>>> (define a 22)
a
>>> a
22
```

```
>>> (define f (lambda (a) (* a a)))
f
>>> (f 7)
49
>>> a
22
```

Local scope can also be created by various versions of the let expression in Scheme. The basic let expression, actually a special form, elaborates the bindings simultaneously and then evaluates the expression expr in the current environment augmented by these bindings.

(let ((id$_1$ val$_1$) … (id$_n$ val$_n$)) expr)

The expression (let ((a 5) (b 8)) (+ a b)) is an abbreviation of the function application ((lambda (a b) (+ a b)) 5 8); both expressions return the value 13. The let expression used to illustrate static scoping in section 8.2 takes the following form in Scheme:

```
>>> (let ((x 5))
          (let ((f (lambda (y) (+ x y))))
               (let ((x 3))
                    (f x))))
8
```

The translation into function applications is not as easy to read.

```
>>> ((lambda (x)
          ((lambda (f)
               ((lambda (x) (f x))
                 3))
           (lambda (y) (+ x y))))
      5)
8
```

Scheme also has a sequential let, called let*, that evaluates the bindings from left to right.

(let* ((a 5) (b (+ a 3))) (* a b)) is equivalent to

```
>>> (let ((a 5)) (let ((b (+ a 3))) (* a b)))
40.
```

Finally, letrec must be used to bind an identifier to a function that calls the identifier—namely, a recursive definition. The following expression defines fact as an identifier local to the expression.

```
>>> (letrec ((fact (lambda (n)
                        (cond  ((zero? n) 1)
                               (else (* n (fact (sub1 n))))))))
        (fact 5))
120
```

See Chapter 10 for an explanation of the meaning of letrec in terms of fixed points.

Proving Correctness in Scheme

Reasoning about the correctness of programs in imperative languages can be a formidable challenge (see Chapter 11).

- Execution depends on the contents of each memory cell (each variable).
- Loops must be executed statically by constructing a loop invariant.
- The progress of the computation is measured by "snapshots" of the state of the computation after every instruction.
- Side effects in programs can make correctness proofs very difficult.

Functional languages are much easier to reason about because of referential transparency: Only those values immediately involved in a function application need to be considered. Programs defined as recursive functions usually can be proved correct by an induction proof. Consider a Scheme function that computes the sum of the squares of a list of integers.

```
(define sumsqrs (lambda (L)
                     (cond ((null?  L)  0)
                           (else  (+  (* (car L) (car L))  (sumsqrs (cdr L)))))
```

Notation: If L is a list, let L_k denote the k^{th} element of L.

Precondition : L is a list of zero or more integers.

Postcondition : (sumsqrs L) = $\sum_{1 \le k \le \text{length}(L)} L_k^2$

Proof of correctness: By induction on the length n of L.

Basis : n = length(L) = 0

Then $\sum_{1 \le k \le \text{length}(L)} L_k^2 = 0$ and (sumsqrs L) returns 0.

Induction step : Suppose that for any list M of length n,

(sumsqrs M) = $\sum_{1 \le k \le \text{length}(M)} M_k^2$.

Let L be a list of length n+1. Note that (cdr L) is a list of length n.

Therefore (sumsqrs L) = L_1^2 + (sumsqrs (cdr L))

$= L_1^2 + \sum_{2 \le k \le \text{length}(L)} L_k^2 = \sum_{1 \le k \le \text{length}(L)} L_k^2$.

Higher-Order Functions

Much of the expressiveness of functional programming comes from treating functions as first-class objects with the same rights as other objects—namely, to be stored in data structures, to be passed as parameters to subprograms, and to be returned as function results.

In Scheme, functions can be bound to identifiers using define and may also be stored in structures:

```
(define fn-list (list  add1  –  (lambda (n) (* n n))))
```

or alternatively

```
(define fn-list  (cons add1
                   (cons  –
                      (cons (lambda (n) (* n n)) '( )))))
```

defines a list of three unary functions.

fn-list returns (#<PROCEDURE add1> #<PROCEDURE – > #<PROCEDURE>).

A Scheme procedure can be defined to apply each of these functions to a number:

```
     (define construction
               (lambda (fl x)
                    (cond  ((null? fl) '( ))
                          (else (cons ((car fl) x) (construction (cdr fl) x))))))
```

so that

(construction fn-list 5) returns (6 –5 25).

The function construction is based on an operation found in FP, a functional language developed by John Backus (see [Backus78]). It illustrates the possibility of passing functions as arguments.

Since functions are first-class objects in Scheme, they may be stored in any sort of structure. It is possible to imagine an application for a stack of functions or even a tree of functions.

Definition : A function is called **higher-order** if it has one or more functions as parameters or returns a function as its result. ∎

Higher-order functions are sometimes called functional forms since they allow the construction of new functions from already defined functions. The expressiveness of functional programming comes from the use of functional forms that allow the development of complex functions from simple functions using abstract patterns—for example, construction defined above. We continue, describing several of the most useful higher-order functions.

- **Composition**

```
(define  compose  (lambda (f g) (lambda (x) (f (g x)))))
(define  inc-sqr  (compose add1 (lambda (n) (* n n))))
(define  sqr-inc  (compose (lambda (n) (* n n)) add1))
```

Note that the two functions inc-sqr and sqr-inc are defined without the use of parameters.

```
(inc-sqr 5) returns 26
(sqr-inc 5) returns 36
```

- **Apply to all**

In Scheme, map is a predefined function that applies a functional argument to all the items in a list. It takes a unary function and a list as arguments and applies the function to each element of the list returning the list of results.

```
(map  add1  '(1 2 3)) returns (2 3 4)
(map  (lambda (n) (* n n))  '(1 2 3)) returns (1 4 9)
(map  (lambda (ls) (cons 'a ls))  '((b c) (a) ( ))) returns  ((a b c) (a a) (a))
```

The function map can be defined as follows:

```
(define map (lambda (proc lst)
                  (if   (null? lst)
                        '( )
                        (cons (proc (car lst)) (map proc (cdr lst))))))
```

- **Reduce**

Higher-order functions are developed by abstracting common patterns from programs. For example, consider the functions that find the sum or the product of a list of numbers:

```
(define sum (lambda (ls)
                  (cond  ((null? ls) 0)
                         (else (+ (car ls) (sum (cdr ls)))))))
(define product (lambda (ls)
                  (cond  ((null? ls) 1)
                         (else (* (car ls) (product (cdr ls)))))))
```

The common pattern can be abstracted as a higher-order function reduce (also called foldright):

```
(define reduce (lambda (proc init ls)
                  (cond  ((null? ls) init)
                         (else (proc (car ls) (reduce proc init (cdr ls)))))))
```

Reduce can be used to compute both the sum and product of a list of numbers.

```
>>> (reduce + 0 '(1 2 3 4 5))
15
>>> (reduce * 1 '(1 2 3 4 5))
120
>>> (reduce concat '( ) '((1 2 3) (4 5) (6 7 8)))
(1 2 3 4 5 6 7 8)
```

Now sum and product can be defined in terms of reduce:

```
(define sum  (lambda (ls) (reduce + 0 ls)))
(define product  (lambda (ls) (reduce * 1 ls)))
```

- **Filter**

  By passing a Boolean function, it is possible to "filter" in only those elements from a list that satisfy the predicate.

```
(define filter (lambda (proc ls)
                 (cond ((null? ls)  '( ))
                       ((proc (car ls)) (cons (car ls) (filter proc (cdr ls))))
                       (else (filter proc (cdr ls))) )))
```

(filter even? '(1 2 3 4 5 6)) returns (2 4 6).

(filter (lambda (n) (> n 3)) '(1 2 3 4 5)) returns (4 5).

Currying

A binary function—for example, + or cons—takes both of its arguments at the same time. For example, (+ a b) will evaluate both a and b so that their values can be passed to the addition operation.

Having a binary function take its arguments one at a time can be an advantage. Such a function is called **curried** after Haskell Curry. (See the discussion of currying in Chapter 5.)

```
(define curried+ (lambda (m) (lambda (n)  (+ m n)) ))
```

Note that if only one argument is supplied to curried+, the result is a function of one argument.

(curried+ 5) returns #<procedure>

((curried+ 5) 8) returns 13

Unary functions can be defined using curried+, as shown below:

```
        (define add2  (curried+ 2))
        (define add5  (curried+ 5))
```

In some functional languages—for example, Standard ML and Miranda—all functions are automatically defined in a curried form. In Scheme, curried functions must be defined explicitly by nested lambda expressions.

- **Curried Map**

```
        (define cmap (lambda (proc)
                          (lambda  (lst)
                              (if  (null? lst)
                                   '( )
                                   (cons (proc (car lst)) ((cmap proc) (cdr lst))))))
```

```
    (cmap add1)  returns  #<procedure>
    ((cmap add1)  '(1 2 3))  returns  (2 3 4)
    ((cmap  (cmap add1))  '((1) (2 3) (4 5 6)))  returns  ((2) (3 4) (5 6 7))
    (((compose cmap cmap)  add1)  '((1) (2 3) (4 5 6))) returns  ((2) (3 4) (5 6 7))
```

The notion of currying can be applied to functions with more than two arguments.

Tail Recursion

One criticism of functional programming centers on the heavy use of recursion that is seen by some critics as overly inefficient. Scheme and some other functional languages have a mechanism whereby implementations optimize certain recursive functions by reducing the storage on the run-time execution stack.

Example : Factorial

```
        (define factorial (lambda (n)
                              (if   (zero? n)
                                    1
                                    (* n (factorial (sub1 n))) )))
```

When (factorial 6) is invoked, activation records are needed for six invocations of the function—namely, (factorial 6) through (factorial 0). Without each of these stack frames, the local values of n—namely, n=6 through n=0—will be lost so that the multiplication at the end cannot be carried out correctly.

At its deepest level of recursion all the information in the expression

```
    (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0)))))))
```

is stored in the run-time execution stack. ∎

Definition : A function is **tail recursive** if its only recursive call is the last action that occurs during any particular invocation of the function. ▮

Example : Factorial with Tail Recursion

```
(define fact (lambda (n)
          (letrec ((fact-help
                    (lambda (prod count)
                       (if  (> count n)
                            prod
                            (fact-help   (* count prod)
                                         (add1 count)) ))))
               (fact-help 1 1))))
```

Note that although fact-help is recursive, there is no need to save its local environment when it calls itself since no computation remains after that call. The result of the recursive call is simply passed on as the result of the current activation.

The execution of (fact 6) proceeds as follows:

```
(fact 6)
   (fact-help 1 1)
   (fact-help 1 2)
   (fact-help 2 3)
   (fact-help 6 4)
   (fact-help 24 5)
   (fact-help 120 6)
   (fact-help 720 7)
```

The final call is the base case, which returns 720 directly. Note that the static scope rules make the value of n visible in the function fact-help. ▮

Scheme is a small, elegant but amazingly powerful programming language. We have been able to present only a few of its features in this overview and have not shown the full range of data types, mutation of data structures (imperative programming in Scheme), object-oriented programming techniques, stream processing, declaring and using macros, or continuations (as briefly discussed in section 9.7). However, we have presented enough concepts so that the reader can write simple Scheme functions and understand the use of Scheme in this text.