Chapter 11 - The Functional Model

The programming language model that is introduced in this chapter is based on the mathematical concept of function, which is a mapping from a domain set to a range set. When this concept is used as a model for programs, the domain is the set of all possible inputs and the range is the set of all possible outputs.

Functions are precisely defined and analyzed. The concept of functional programming is introduced and languages that implement that paradigm are described. A brief review of the development of functional languages is presented and the Scheme programming language is described as an implementation of the functional model. Finally, functional programming languages are compared with other models.

11.1 Introduction to the Functional Model

A function has three basic components: domain, range, and definition. The domain is the set of objects to which the function can be applied, the range is a set containing all objects that can result from an application of the function, and the definition is a specification of how a range element is determined from a domain element. A fourth, optional, component of a function is its name.

For example, if we wish to define a function called \textit{double}, we can specify its four components as follows:

\begin{itemize}
  \item Domain : set of integers
  \item Range : set of integers
  \item Definition : \( x + x \) where \( x \) is an element from the domain
  \item Name : double
\end{itemize}

In mathematical notation, we would write this function definition as

\[ \text{double}(x) = x + x \]

and specify the domain and range by

\[ \text{double} : \text{integer} \rightarrow \text{integer} \]

The mathematical notation for the application of a function to a specific domain value is the name of the function followed by the domain value enclosed in parentheses. For example, function \textit{double} applied to 2 is written as

\[ \text{double}(2) \]

Several points need to be made about this notation. First, it makes naming a function a requirement, because the name is a required part of the function's application. Second, note that the function definition includes the operator \textit{plus}, which is in reality a function itself. A more consistent functional notation for \( x+y \) would be \( +(x,y) \) to indicate that + is a function whose domain is the set of integer pairs and where the range is the set of integers.

For our future discussion of functions, we introduce a notation that differs from the above traditional mathematical one for the definition and application of functions. This form of expressing functions is known as the \textbf{lambda expression}, after the Greek letter used as a part of the notation. It was developed by Alonzo Church (1941).

The new form of function definition is

\[ [<\text{function name}>]= \lambda<\text{list of domain element names}>,<\text{definition}> \]

For our function \textit{double}, the definition is

\[ \text{double} = \lambda x. x+x \]

The fact that the name specification is optional permits the definition of a function without assigning that function a
name.
  In addition, the application of a function now has a completely different form from its definition. The lambda expression form for function application is

  \langle \text{function specifier} \rangle : \langle \text{domain value} \rangle

The function specifier could be either a name or the definition of a function. Given the preceding definition of \textit{double}, this function could be applied to domain value 2 by

  double : 2

or equivalently by

  \lambda x. x+x : 2

As an example of a function with a composite domain, consider the \textit{max} function, which can be defined by

  \begin{align*}
  \text{max} : \text{integer} \times \text{integer} & \rightarrow \text{integer} \\
  \text{max} & \equiv \lambda x, y. \text{if } x>y \text{ then } x \text{ else } y
  \end{align*}

Note that, for our purposes, the definition can be specified in any form here as long as any domain tuple maps into a unique range value. The application of the function \textit{max} is then written, in lambda format, as

  \text{max} : \langle 4, 2 \rangle

where the angle brackets are used to enclose the components of a multicomponent domain.

The definition of functions can frequently include the application of other functions. For example, the absolute value function can be defined as

  \text{abs} \equiv \lambda x. \text{max:}\langle x, -x \rangle

The mechanics of the application of a function are straightforward. Each name appearing in the domain list of the definition is bound to the positionally corresponding value in the function application. These names are referred to as \textit{bound variables}. In the case of the \textit{abs} function just defined, its application specified by

  \text{abs} : 6

results in 6 replacing \(x\) in the definition of \textit{abs}, yielding

  \text{max} : \langle 6, -6 \rangle

This application of \textit{max} would then result in \(x\) being replaced by 6 and \(y\) by -6 in the \textit{max} definition, giving

  \text{if } 6>-6 \text{ then } 6 \text{ else } -6

which would evaluate to the value 6. We can more compactly represent this application by

  \begin{align*}
  \text{abs} : 6 & \Rightarrow \\
  \text{max} : \langle 6, -6 \rangle & \Rightarrow \\
  \text{if } 6>-6 \text{ then } 6 \text{ else } -6 & \Rightarrow
  \end{align*}
One further feature is the ability to compose two functions to form a new function. For example, suppose we wish to construct a function that returns the absolute value of twice the given domain value. In mathematical notation, we can write

\[ f(x) = \text{abs}(\text{double}(x)) \]

In our lambda expression notation, we define such function composition by a new operator. This operator defines a new function from two previously defined functions through composition. The function \( f \) defined by the composition of \( \text{abs} \) and \( \text{double} \) is defined in lambda notation by

\[ f \equiv \text{abs} \circ \text{double} \]

This means that an application of \( f \) is equivalent to successive applications of \( \text{double} \) and \( \text{abs} \).

To see how composition behaves during its application, study the application description below:

\[
\begin{align*}
  f &: -3 \quad \Rightarrow \\
  \text{abs} &: \text{double} : -3 \quad \Rightarrow \\
  \text{abs} &: -3 + -3 \quad \Rightarrow \\
  \text{abs} &: -6 \quad \Rightarrow \\
  \text{if} &: -6 > 6 \text{ then } -6 \text{ else } 6 \quad \Rightarrow \\
  6
\end{align*}
\]

Composition is an example of what we will call a functional form, namely, a function whose domain is a tuple (possibly a singleton) of functions and whose range is also a function. For example, composition takes two functions as domain values and results in a third function, which is the composition of the original two.

### 11.2 Functional programming

To a certain degree all programming may be thought of as functional in nature: that is, programs can be considered as representations of functions. The input given to the program corresponds to the functional parameters, whereas the output corresponds to the result of the function. Functional programming carries this idea to the most fundamental level of program construction.

Whereas imperative programming is based on the sequential modification of some internal machine state or store, functional programming builds programs from other programs, always considering the programs as black boxes with no consideration for the sequence of activities that must be performed or for the progression of internal states of the data store during the computation.

The key property of a functional program is referential transparency. This refers to the ability to call a function without producing side effects—that is, without changing the internal state of the computations. A function that exhibits referential transparency depends only upon its parameters and modifies only its return value. In languages such as C++, the use of nonlocal variables or file operations destroys referential transparency.

The main value of referential transparency is in enhancing our ability to reason about a program, either formally or informally, because the complete description of the function’s activity is specified by its definition, with no other intervening factors.

A second property of functional programming is the treatment of functions as first class objects. This property means that functions are treated like any other object in the language. In particular, it means that functions can be used as parameters and return values of other functions.

### 11.3 Functional programming languages

Languages of the functional programming model have six majors components.
1. A set of primitive functions--Primitive functions are basic functions that are built into the language and that can be used as the fundamental building blocks for all functions that can be constructed.

2. A set of functional forms--Functional forms are functions that accept functions as parameters. If functions are first-class objects in a language, the set of built-in functions might include a set of built-in functional forms. Otherwise, functional forms might be provided separately or given as operators instead of functions. Functional forms are useful for constructing new functions from functions that are already defined.

3. The application operator--The application operator is a special functional form that takes as parameters a function and a set of parameter values and returns the result of applying the function to the parameters.

4. A set of data objects--Data objects in a functional language typically consist of some atomic type of object and the ability to construct aggregate objects from atoms and other aggregate objects. Languages use different models for constructing aggregates: FP uses sequences, Lisp and Scheme use lists, APL uses arrays.

5. Binding of names to functions--This binding is the only form of permanent binding provided in a purely functional language. Most languages compromise on the issue of referential transparency, however, and provide some form for binding names to data objects as well.

6. Implicit storage management--Because functional languages do not provide facilities for directly modifying the state of the storage for a computation, the management of the store must be handled implicitly. This typically includes implicit dynamic storage allocation and garbage collection.

An additional issue that frequently arises in functional languages is lazy function evaluation. This refers to a strategy that eliminates unnecessary evaluations of functions and includes two substrategies: (1) postponing evaluation of a function until it is needed, and (2) eliminating the reevaluation of the same function more than once.

As an illustration of these two strategies, consider the following construction:

   if g(f(x)) then m(f(x)) else n(f(x))

This is an if–then construct and if it is evaluated strictly, it requires six function evaluations. Lazy evaluation, however, reduces this to only two function evaluations, because f needs to be evaluated only once (thanks to referential transparency) and only one of the two functions m and n need to be evaluated during a given evaluation of this conditional function.

11.4 History of functional languages

A detailed discussion of the evolution of functional languages is given in Hudak (1989). We briefly summarize this evolution here. Lisp was the first functional languages, developed by John McCarthy in the late 1950s for artificial intelligence applications. Dialects of List like Logo and Scheme, have also become heavily used. Scheme will be discussed in detail in this chapter.

APL, although not designed to be a functional language, has some strong functional characteristics. APL contains both the assignment statement and statement sequencing as fundamental non-functional features, but it includes enough powerful operators to permit functional definition of programs as well. It differs from Lisp in its introduction of an expanded alphabet to express operators, its use of infix rather than prefix notation, and its use of arrays rather than lists as the fundamental data structure. Although it contains several functional forms in its language definition, APL does not treat functions as first-class objects and does not have referential transparency.

In his 1978 Turing Award lecture, John Backus (1978) defined the class of functional programming languages called FP. The purpose of his paper was to define and advocate functional programming. FP followed the APL model of introducing an extensive set of functional forms.
The language Standard ML, or SML, was designed by a team in the United Kingdom in the mid 1980s. It is described in Milner (1984) and Wikstrom (1988). Combining the features of two earlier languages, ML and Hope, its main feature is its strong type facility. This permits the definition of abstract data types as well as overloading and polymorphism. SML is described in Chapter 13.

Several more recent functional programming languages are based on the \( \lambda \)-calculus of expressions. The most prominent of these are Miranda and Haskell.

### 11.5 Scheme - A Functional-Oriented Language

The programming language LISP adheres more closely to the functional model than any other language in general use. LISP was developed by John McCarthy in 1960 to facilitate work on the list data structure. The language was designed so that programs in LISP are lists themselves and it is based on the functional model, complete with lambda expressions and recursion. Over the years, LISP has been the favorite language among researchers in the field of artificial intelligence.

There is no single established standard version of LISP, but one version that has emerged as important in the field of education is Scheme. Scheme was developed at MIT by Sussman and Steele (1975) and is an improvement over earlier dialects of LISP in that it uses static scope and treats functions as first class objects. Scheme is the dialect of LISP that we discuss in this section, where we study features of Scheme in comparison to the functional model. We also introduce some features of Scheme that lie outside of that functional model.

### 11.6 Basic Components of Scheme

#### 11.6.1 Objects and Evaluations

Scheme permits two types of objects, atoms and lists. **Atoms** are represented by strings of nonblank characters. Those atoms that are represented by numeric characters are called **numeric atoms** and admit to the application of a set of numeric functions. Other atoms may be enclosed in double quotes and are considered string atoms. Atoms that are neither numeric nor string are called **identifier atoms**. Sample atoms are

- \( 28 \)
- \(-14.292 \)
- "A string"
- \( x \)
- \( \text{anAtom} \)

A **list** is represented by a sequence of atoms and lists separated by blanks and enclosed in parentheses. Examples of Scheme lists are

- \( ( x \ y \ z ) \)
- \( (+ 14 12) \)
- \( (x \ (a \ b \ c) \ ()) \)

The special list \( () \) that contains no elements is called the empty list. The empty list is considered both an atom and a list. It is alternatively referred to as \( \text{nil} \).

Every expression in Scheme specifies an evaluation. In this text, we indicate the result of expression evaluation by printing the expression on one line with the result of the evaluation of the expression indented on the following line.

The rule for evaluating an expression is as follows:

1. If the expression is a numeric or string atom, it evaluates to itself--for example
2. An atom that is an identifier is bound to a function or a numeric or string atom and evaluates to the value to which it is bound. For example, assuming the appropriate bindings have been made via techniques to be described later, the following are example results of identifier evaluations:

```
X
```
```
Name
```
```
"Joseph"
```
```
sort
```
```
#PROCEDURE SORT
```

3. A list is treated as a function evaluation with the first element representing the function and the remaining elements of the list serving as its parameters. We will examine this evaluation later.

### 11.6.2 Implementation Model

The **cell model** is commonly used to view the implementation of lists in Scheme. In the cell model, a list is represented by a linked list of cells. The data component of each cell is a pointer to the corresponding atom or list, and the pointer element points to the next element in the list. The last element in a list points to `nil`. Figure 11.1 shows several sample lists represented by the cell model. Using conventional Scheme notation, a pointer to `nil` is indicated by a diagonal line in our diagram.

Scheme uses special terminology for the two components of each cell. The data part is known as the **CAR** component and the pointer part is called the **CDR** component. These terms are a consequence of the historical names used for the registers where these components were stored in the original implementation of LISP. CAR stands for "Contents of Accumulator Register" and CDR stands for "Contents of Decrement Register". In actuality, the CAR is a pointer to the first element of the list and the CDR is a pointer to the cells that represent the remainder of the list after the first element is removed.

### 11.6.3 Function Application

In order to enforce the property that functions are represented by lists, Scheme requires the special notation for expressing function applications known as the **S-expression**. Its general form is

```
(<function-name> <first-parameter> ... <last-parameter>)
```

In words, a function application is represented by a list whose first element is an atom expressing the function's name and whose remaining elements are atoms or lists representing the parameters to the function. For example, `6` is added to `9` by the S-expression

```
(+ 6 9)
```
```
15
```

More complex evaluations can be specified by groupings of function applications. For example,

```
(+ (* 3 4) (* 6 7))
```
This syntax for specifying function application is called prefix form as the function name precedes the list of parameters. In normal arithmetic operation syntax, operator functions like + are used in infix form, with the operator between its two operands.

**Figure 11.1 Cell model representation of lists**

```
(a b c)
```

```
(a (b c (d) e) )
```

```
((a b) nil)
```

### 11.6.4 Built-in Functions

Functions in Scheme are either built-in or user-defined. We discuss the way the user can define functions later. In the present section we describe several of the most important built-in functions.
Because the Scheme interpreter considers every S-expression to be a function to be evaluated, if Scheme wishes to consider an S-expression as data and prevent its evaluation, this must be specified with a special function called quote. In other words, Scheme considers a list to be a function evaluation unless told otherwise by the presence of a quote function. For example

\[
(\text{quote } (+ 2 3))
\]

\[
(+ 2 3)
\]

indicates to the Scheme interpreter that the list

\[
(+ 2 3)
\]

is to be considered as a three element list rather than be evaluated and regarded as the atomic element 5. Since atoms are considered as elements to be evaluated in a way that will be explained later, the quote function may also be applied to them to inhibit their evaluation—for example:

\[
(\text{quote } a)
\]

\[
a
\]

Due to the frequency with which the function \text{quote} is applied, it is often abbreviated by a single quote mark, shorthand notation allowed by Scheme. For example, the preceding expressions can be equivalently written

\[
'(+ 2 3)
\]

\[
(+ 2 3)
\]

\[
'a
\]

\[
a
\]

The quote function is applied to a function’s actual parameters when they are to be considered as data rather than other functions to be evaluated.

Two other commonly used built-in Scheme functions are \text{car} and \text{cdr}, named after the two components of the cell in the cell model. These functions both take a list as their only parameter, the \text{car} returning the first element of that list and the \text{cdr} returning the list parameter with the first element removed. In other words, the \text{car} returns the element to which the CAR component points and \text{cdr} returns the element to which the CDR component points—for example,

\[
(\text{car } '(a b c))
\]

\[
a
\]

Applying the \text{cdr} function to the same list gives

\[
(\text{cdr } '(a b c))
\]

\[
(b c)
\]

Note that this result of the \text{car} is an atom, because the first element of the list \((a b c)\) is an atom. The result of the \text{cdr} is always a list. Neither the \text{car} nor the \text{cdr} function is defined when its parameter is an atom or \text{nil}. Further examples of applications of \text{car} and \text{cdr} are shown here and are numbered for convenience:

1. \((\text{car } '((a b) c d))\)

\[
(a b)
\]

2. \((\text{car } '(a))\)

\[
a
\]

3. \((\text{cdr } '((a b) c))\)

\[
(b c)
\]
4. (cdr 'a))
   nil
5. (car (cdr 'a b c)))
   b
6. (car 'c (cdr a b c)))
   cdr

The last two examples are of special interest. In example 5, because the list to which \texttt{car} is applied is not quoted, it is evaluated before the \texttt{car} function is applied. Therefore, this expression is equivalent to

\begin{verbatim}
(car 'b c))
\end{verbatim}

because \((b c)\) is the \texttt{cdr} of \((a b c)\). In example 6, the \texttt{car} function is applied to the two element list

\begin{verbatim}
(cdr a b c))
\end{verbatim}

because the preceding quote inhibits evaluation. The \texttt{car} of this list is the atom \texttt{cdr}.

The \texttt{car} and \texttt{cdr} functions are the two fundamental list functions for taking a list apart. The \texttt{cons} function is the fundamental function for constructing a list. It takes two parameters, the first becoming the \texttt{car} of the result and the second the \texttt{cdr}--for example,

\begin{verbatim}
(cons 'a 'b c))
\end{verbatim}

\begin{verbatim}
(a b c)
\end{verbatim}

In contrast, if the first parameter is a list, then we have

\begin{verbatim}
(cons 'a (b c))
\end{verbatim}

\begin{verbatim}
((a) b c)
\end{verbatim}

The inverse relationship of \texttt{cons} with \texttt{car} and \texttt{cdr} is illustrated by the rule that

\begin{verbatim}
(cons (car X) (cdr X))
\end{verbatim}

is \(X\) for any nonempty list \(X\).

Scheme recognizes the atom \#t to have the meaning "true" when used in the context of logical results. The meaning "false" is associated with \#f.

In this context, a \texttt{predicate} is a function whose result is always either \#t or \#f. For example, built-in function \texttt{atom?} returns \#t if its parameter is an atom and \#f if it is not. Consider the following:

\begin{verbatim}
(atom? 'a)
\end{verbatim}

\begin{verbatim}
#t
\end{verbatim}

\begin{verbatim}
(atom? '(a))
\end{verbatim}

\begin{verbatim}
#f
\end{verbatim}

\begin{verbatim}
(atom? '(a b))
\end{verbatim}

\begin{verbatim}
#f
\end{verbatim}

Another commonly used predicate is \texttt{equal?}, which tests its two parameters for equality, where the parameters can be atoms or lists. In addition, comparison predicates exist that apply to numeric atoms, namely \(=\), \(>\), and \(<\).

One further construct is used to abbreviate compositions of \texttt{car} and \texttt{cdr} applications--for example,
(cdr (cdr (car '((a b c)(d e)))))

can be abbreviated as

(cddar '((a b c)(d e)))

In general, a sequence of compositions of car and cdr applications can be abbreviated into a single function. There are two ways to represent conditional evaluation within Scheme. In adherence to the functional nature of the language, both are expressed as functions.

The first takes the general form of an if-then-else by application of the if function. Its general form is

(if <predicate>
  <expression-1>
  <expression-2>)

If the predicate evaluates to anything but #f, the if function result is the evaluation of expression-1. If the predicate evaluates to #f, the if function result is the evaluation of expression-2. For example,

(if (= n 0)
  0
  (/ 1 n))

evaluates to 0 if n is zero and to 1/n if n is not zero. Observe that a non-nil parameter that is neither #f nor #t produces the same result as if it were #t.

A second conditional is the cond function, which has the form

(cond (<test-1> <result-1>)
  (<test-2> <result-2>)
  ...
  (<test-n> <result-n>)
)

The cond function evaluates the test clauses in order until one is found that is not #f. The value of the corresponding result clause is then returned as the value of the cond function. If all test clauses are #f, the value of the cond function is nil. In this sense it acts like an if-elsif-elsif-...-endif type of construct. Illustrations of the application of cond will be shown in later examples.

11.6.5 Function Definition

In the previous section we introduced a number of built-in Scheme functions. In this section we will discuss how to construct user-defined functions in Scheme. This construction is done in the typical functional manner through the application of the define function. The general form of this function defining function is

(define (<function-name> (<parameter-1> ... <parameter-n>)
  <function-body>)
)

Accordingly, function define takes two parameters. The first is a list with first element being the function identifier
and the remaining elements being atomic formal parameter names. The second element is the expression that is to be evaluated when the function is called. The atomic function name is returned as the result of the function define, but the primary result of the define application is the side effect that binds the function identifier with the function definition provided in the body. Formal parameter names are used like local variables in the function body.

Consider the following example:

```
(define (length list)
    (cond ((atom? list) 0)
          ((null? list) 0)
          (else (+ 1 (length (cdr list))))
    )
)
```

This function counts the number of elements in its formal parameter. Note the use of the conditional and recursion in the body of the function. The function length can now be called by the form

```
(length '(a b (c d)))
```

3

Functions in Scheme can also be defined without binding them to a name. This is done through the use of lambda expressions as defined in Section 11.1. The general form is

```
(lambda (<parameter-1> ... <parameter-n>)
  <function-body>
)
```

This expression defines the specified function but, rather than binding it to a name, makes the function available for an application without naming it. For example, the preceding call to length could be replaced by the lambda expression application shown here:

```
((lambda (list)
    (cond ((atom? list) 0)
          ((null? list) 0)
          (else (+ 1 (length (cdr list))))
    )
  )
'(a b (c d))
)
```

3

Here, the result of the evaluation of the lambda function is a function that is then applied to its single actual parameter (a b (c d)).

The parameters within function calls act as parameters implemented by copy. Therefore, if we are given the function f defined by

```
(define (f test)
    ((define test () )
     (null? test)
    )
)
```

the nil value to which the formal parameter is set has no effect on the corresponding actual parameter in the calling
environment. The function always returns `nil`, however. This is illustrated by the following sequence of evaluations:

```
(define x 'a)
x
(f x)
#t
x
a
```

Note that the formal parameter `test` does change during the evaluation of function `f`, but the associated actual parameter `x` is unaffected. In addition, all nonparametric variables used in a function definition are considered global to the defining environment in Scheme. Some LISP dialects use dynamic scoping by allowing the function to inherit the variable name bindings of the calling environment.

The function `f` defined above illustrates another important point about Scheme function definitions. A function body may contain any number of evaluations. In the situation where there is more than one, only the result of the last evaluation is returned as the value of the function. Also, note that this function introduces a non-functional capability of Scheme inasmuch as the two functions are evaluated sequentially. We’ll look at this further in Section 11.8.

### 11.7 Scheme Examples

The best way to understand the true power of the Scheme language is through the use of examples.

#### 11.7.1 Example 1 - Factorial

A simple function to evaluate factorial in Scheme is defined by

```
(define (factorial x)
  (cond ((< x 0) ()
      ((= x 0) 1)
      (else (* x (factorial (- x 1))))
      ))
)
```

Here a negative parameter is considered illegal and hence returns `nil`. The function is called by a form such as

```
(factorial 6)
720
```

#### 11.7.2 Example 2 - Quicksort

Our quicksort function in Scheme makes use of two functions `keep_le` and `keep_gt`. Both of these functions accept two parameters, the first of which is a numeric atom and the second of which is a list of numeric atoms. The function `keep_le` returns a list of all elements in the second parameter that are less than or equal to the first parameter. The function `keep_gt` performs similarly, but it returns a list of elements greater than the first parameter. These two functions are defined in Figure 11.2.

This same figure also contains the definition of the `quicksort` function. The built-in function `append`, which we have not discussed, is used to concatenate two lists into a single list.
Figure 11.2 Quicksort in Scheme

(define (keep_some f)
  (lambda (x slist)
    (cond ((null? slist)      ()
          ((f (car slist) x) (cons (car slist)
                      ((keep_some f) x (cdr slist))))
          ( else               ((keep_some f) x (cdr slist))))
    )
  )
)

(define (leq x y) (if (> x y) #f #t))

(define keep_gt (keep_some >))

(define keep_le (keep_some leq))

(define (quicksort slist)
  (if (null? slist)
      ()
    (append (quicksort (keep_le (car slist) (cdr slist)))
            (cons (car slist) (quicksort (keep_gt (car slist) (cdr slist))))))
  )
)

11.7.3 Example 3 - Stacks

To implement stacks in Scheme, we represent the stack by a list, with the first element of the list being the top element of the stack. The three fundamental stack functions are then defined by

empty : stack → #t if stack is empty
       #f if stack is not empty

push : stack,element → newstack where newstack is stack with element on top

pop : stack → (element,newstack) where element is the top and newstack with top removed

The function pop returns a list consisting of the element popped off of the stack and the resulting stack after the element is popped. This action is necessary, because a function can return only a single value. We get around this limitation by returning the two values as the components of a single list.

Our function definitions are found in Figure 11.3. The empty and push functions are direct applications of the null? and cons built-in functions. The pop function is constructed via a conditional, which returns nil if the stack is empty to indicate an improper pop operation. If the stack is not empty, pop is the application of a built-in Scheme function called list. This function is similar in operation to cons, except that it makes a list with its first parameter as the first element and its second parameter as the second element, and so on, for any number of parameters provided. To illustrate, consider the following examples:

(list 'a 'b c))
(a (b c))
(cons 'a '(b c))
(a b c)

Figure 11.3 Stack Implementation in Scheme

(define (empty? stack)
  (null? stack))
(define (push stack element)
  (cons element stack))
(define (pop stack)
  (if ((empty? stack)
       nil
       (list (car stack) (cdr stack)))


11.7.4 Example 4 - Attribute Lists

We assume that we have objects that are described by lists of pairs, the first element of each pair being the name of an attribute and the second being the value of that attribute associated with the given object. Although Scheme has built-in functions that implement the insertion and retrieval of attribute values, we write our own here for illustrative purposes. As an example, a student object might be represented by the list

((lastname Smith) (firstname Sam) (class FR) (sex M))

We provide three functions for use with such attribute lists specified by

  addattr : attr-list,attr-name,attr-value → new-attr-list
  adds a name/value pair to the attribute list
  remattr : attr-list,attr-name → new-attr-list
  removes the first pair with attr-name from attr-list
  setattr : attr-list,attr-name → attr-value
  retrieve the value paired with attr-name in attr-list

The preceding functions are defined in Scheme in Figure 11.4. Both remattr and setattr illustrate the use of recursion to perform a list search.

Figure 11.4 Attribute Lists in Scheme

(define (addattr alist aname avalue)
  (cons (list aname avalue) alist))
(define (remattr alist aname)
  (cond ((null? alist)      ()
         (equal? (caar alist) aname) (cdr alist)))
(define (getattr alist aname)
  (cond ((null? alist)                ()))
       ((equal? (caar alist) aname) (cadar alist))
       (else                       (getattr (cdr alist) aname)))

11.8 Functional Forms in Scheme

Within Scheme, functions are treated as first-class objects, which means they can be used as parameters to functions and returned as the value of a function. This gives Scheme the power to express functional forms. Some common functional forms are defined here as illustrations.

The composition of two functions is the function resulting from the successive application of the two functions. In Scheme, a composition functional form can be defined by

\[
\text{(define (composition f g)}
\text{  (lambda (x) \text{ (f (g x))})})
\]

This function accepts two functions, \( f \) and \( g \), as parameters, and the result is a third function defined by the lambda expression, which is the composition of \( f \) and \( g \). This new function can then be applied as follows:

\[
((\text{composition car car}) '((a b) c))
\]

\[
(a, c)
\]

\[
((\text{composition cdr car}) '((a b) c))
\]

\[
(b)
\]

Another example is the apply-to-all functional form which is defined in Scheme by

\[
\text{(define (apply-to-all f)}
\text{  (lambda (list)}
\text{    (if (null? list)
\text{      (nil)
\text{      (cons (f (car list)) ((apply-to-all f) (cdr list)))))})})
\]

This function constructs a new function from its single parameter that applies the parameter to each element in a list, resulting in the list of all of these results. Some applications of this function are given below:

\[
((\text{apply-to-all car}) '((a b) (c d) (e f)))
\]

\[
(a, c, e)
\]

\[
((\text{apply-to-all atom?}) '((a b) (c d) (e f)))
\]

\[
(#t #f #f #t)
\]

A third function form is known as construction. It accepts as parameters a list of one-parameter functions and a value. The result is a list of the results of applying each function in the list to the second parameter. An example appli-
cation of this function is

\[\text{(construction (car cdr atom?) '(a b c))}\]
\[\text{a (b c) #f}\]

The construction functional form is defined in Figure 11.6.

**Figure 11.6 Construction functional form in Scheme**

```scheme
(define (construction flist)
  (lambda (x) (if (null? flist)
                  ()
                (cons (eval (cons (car flist) (list (quote x))) user-initial-environment)
                      ((construction (cdr flist)) x)))))
)
```

### 11.9 Non-functional features of Scheme

Scheme departs from the functional model established by FP in two fundamental ways:

1. The use of the imperative model concept of variable, permitting names to be bound to values during a function evaluation.
2. The ability of Scheme to specify the sequential execution of functions as illustrated by the `let*` function.

The first of these departures is found in the ability to bind identifiers to values, permitting side-effects. This results in functions whose evaluations may be different when different bindings hold. Therefore, all functions that establish such bindings are in violation of the strict functional model.

The binding of names to values occurs in Scheme through the `let` function. The general format of this function is

```scheme
(let ((name-1 expression-1)
       (name-2 expression-2)
       ...
       (name-n expression-n))
expression-n+1)
```

This function first binds each specified name to the value of its corresponding expression. The value of the function is then the result of the evaluation of the final expression with the appropriate values substituted for the names.

Consider the following example:

```scheme
(let ((n 2)
       (m 3))
  (+ n m))
```

The value of this expression is 5.

The `let` function can be used to avoid redundant function evaluations. For example, the function call

```scheme
(* (+ 2 4) (+ 2 4))
```

results in the same addition occurring twice. This can be reduced to a single evaluation by using
(let ((sum (+ 2 4))
    (* sum sum))
  
The \texttt{let} assumes that all binding are done simultaneously. In other words, the binding made in one expression cannot be used in a later expression. This dependence can be enforced, however, with the \texttt{let*} function. In the case of \texttt{let*}, the bindings occur sequentially and earlier bindings can be used in later ones. The difference between \texttt{let} and \texttt{let*} is illustrated in the following:

\begin{verbatim}
(define a 1)
a
(let ((a 2)
      (b a))
  (+ a b))
3
(let* ((a 2)
       (b a))
  (+ a b))
4
\end{verbatim}

The above calls to \texttt{let} and \texttt{let*} illustrate sequential execution as it is permitted in Scheme. As an additional example of sequential execution, let us modify function \texttt{getattr} of Figure 11.4 to retrieve a list of all attribute values associated with a given attribute name, assuming that several name/value pairs could have the same associated name part--for example,

\begin{verbatim}
((lastname,Smith) (firstname Sam) (brother Joe) (brother Charles))
\end{verbatim}

A retrieval for attribute name "brother" applied with this list as an actual parameter, should result in

\begin{verbatim}
(Joe Charles)
\end{verbatim}

The function \texttt{getall}--defined in Figure 11.5--accomplishes this through the use of sequential function applications. The \texttt{let} function is used here, employing local binding of identifiers.

Function \texttt{getall} uses two local variables, \texttt{rest} and \texttt{result}. Inside the \texttt{let*} two functions are applied sequentially to calculate these variables. The result of the function is the \texttt{cons} of these two.

\textbf{Figure 11.5 Function getall in Scheme}

\begin{verbatim}
(define (getall alist aname)
  (let* ((rest (if (null? alist)
                 ()
                 (getall (cdr alist) aname)))
         (result (if (equal? (caar alist) aname)
                    (cadar alist)
                    () )))
    (cons result rest))
)
\end{verbatim}

\textbf{11.10 ML -- A Typed Functional Language}
A different approach to the implementation of the functional model is represented by the language ML. ML stands for meta language and was developed at the University of Edinburgh by a team headed by Robin Milner. Its original purpose was to find and perform proofs in a formal logical system. The full definition of the original language is given by Gordon et al. (1979).

ML proved to be useful as a general-purpose language and was developed as such by the team at Edinburgh. A standardization effort took place in the early 1980s, incorporating some of the ideas from the functional language HOPE, a language developed by Burstall (1980). This resulted in the definition of Standard ML, or SML. This definition was originally published by Milner, et al. (1990). Standard ML is the version of the language that we describe in this chapter.

11.11 Features of ML

11.11.1 Types

A primary difference between ML and Scheme is the former's use of strict typing. All items in ML are bound to a type and that binding persists for the life of the item. Type is so important in ML that it is reported as a part of each function application. As with Scheme, we indicate the results of function evaluations by showing the ML syntax followed by the report returned by ML on a separate line, indented and in italics. For example, a simple function evaluation in ML is

5 + 2;
val it = 7 : int

This indicates that the application of the built-in infix function + results in a value 7 of type int that is stored in a scratch variable called it, the recipient of all function evaluations not explicitly assigned elsewhere. All statements are terminated by a semicolon, as indicated by this example.

We contrast the above function call to the following:

5.0 + 2.0;
val it = 7.0 : real

The difference here is that the + operator, which is overloaded in ML, is taken to be real addition because the operands are expressed as real constants.

ML does not permit the mixture of types, which requires type conversion. When this is attempted in ML, the result looks like the following:

5.0 + 2;
Error: operator and operand don't agree

All functions are strictly typed in both their parameters and results. A function is defined by the keyword fun—for example,

fun double(x) = 2*x;
val double = fn : int -> int

This says that the identifier double has been bound to a function that takes one int parameter and results in a single int return value. The types associated with this function are inferred by ML, because it is nowhere stated that either the parameter or the return types are int. The inference was made in this case from the presence of the int constant 2 in the definition. Consider, in contrast, the function definition
fun double(x) = x + x;
  Error: overloaded variable cannot be resolved: +

Here there is no clue for ML to use in determining the type of the function, because the operator + applies to both int and real operands. The specification of any single element may be enough for ML to infer correctly all types in the function definition—for example,

fun double(x:int) = x + x;
val double = fn: int -> int

Here ML is able to determine the type of the result, because the type of the parameter is specified. Similarly, specifying the type of the result completely specifies all types for the function, as in

fun double(x):int = x + x;
val double = fn: int -> int

Functions that are user-defined can be applied by writing their names, followed by the value or values that specify their parameters. The function double, defined previously, can therefore be applied in the following ways:

double 5;
  val it = 10 : int
double(5);
  val it = 10 : int
double(5.0);
  Error: operator and operand don't agree

In the last case, ML objects at being given a parameter of incorrect type. Functions can be applied to function results as well—for example,

double(double(3));
  val it = 12 : int

There are four atomic types in ML: int, real, bool, and string. Common operators are provided with each of these types.

Identifiers can be bound to values through statements beginning with the keyword val—for example,

val x = 2;
  val x = 2 : int
val y = 2*x;
  val y = 4 : int
val z = x-y;
  val z = ~2 : int

Note that although ML is very strict about typing, an identifier is not bound to a type, but rather to a value (plus its associated type), and the rebinding of the same identifier to a value of a different type is allowed as in

val x = 2;
  val x = 2 : int
val x = 2.0;
  val x = 2.0 : real
val x = "a string";
val x = "a string" : string

We have seen that the binding of an identifier to a function is accomplished by use of the
fun keyword. An alternative form uses the val keyword to bind the identifier to a value that is a
function, which is one way that functions are treated as first-class values in ML—for example,

val double = fun x : int => x + x;
val double = fn : int -> int

The use of the fun keyword to indicate a value is a function is identical in concept to the λ nota-
tion. It permits the application of a function without the function being named. As an example, consider

(fun x:int => x+x)(2);
val it = 4 : int

Functions that require several alternate specifications can be defined in two ways in ML. Consider factorial, for example, which can be written

fun factorial(n) = if n = 0 then 1
else n*factorial(n-1);
val factorial = fn : int -> int

This form of function specification permits a sequence of alternative conditions, each followed by
its corresponding return value, given in an if-else if-...-else form. If there are more
than two alternatives, they might be specified as follows:

fun sign(x) = if x>0 then "+"
    else if x=0 then "0"
    else "-";
val sign = fn : int -> string

Another way of specifying alternative definitions is by specifying parameters that match
patterns. In this manner, factorial could be written

fun factorial(0) = 1
| factorial(n) = n*factorial(n-1);
val fact = fn : int -> int

This definition searches alternative patterns for parameters until one is found that matches the
parameters in the actual function application. The same approach can be used to generate the
sign function:

fun sign(0) = "0"
| sign(1) = "1"
| sign(~1) = "-"
| sign(x) = sign(x div abs(x));
val sign = fn : int -> string

This last example also illustrates that ~ is used for specifying negative numbers. The built-in func-
tions div and abs are used to divide integers and compute the integer absolute value.
Thus far we have avoided functions that accept more than one parameter, but these functions are handled easily in ML. Consider the following function, which returns a string that is a repetition of its first parameter the number of times indicated by its second parameter.

\[
\text{fun repeat}(s,n) = \begin{cases} 
  \text{""} & \text{if } n=0 \\
  s \text{^ repeat}(s,n-1) & \text{else}
\end{cases}
\]

\[
\text{val repeat = fn : string * int -> string}
\]

The domain of the function is listed as \text{string * int}. This notation indicates that it is the Cartesian product of the two types \text{string} and \text{int}. The string operator \text{^} is used for concatenation. The above function can be applied to a pair of values as indicated below.

\[
\text{repeat("ab",5)};
\]

\[
\text{val it = "ababababab" : string}
\]

**11.11.2 Lists**

List constants in ML are enclosed in brackets and elements are separated by commas. Using that notation, we illustrate the fact that lists must be homogeneous by the following sequence of examples:

\[
[1,2];
\]

\[
\text{val it = [1,2] : int list}
\]

\[
[1,2.0];
\]

\[
\text{Error: operator and operand don't agree}
\]

\[
[1,2,3];
\]

\[
\text{val it = [1,2,3] : int list}
\]

\[
[1,[2,3]];\]

\[
\text{Error: operator and operand don't agree}
\]

\[
[[1],[2,3]];\]

\[
\text{val it = [[1],[2,3]] : int list list}
\]

These examples illustrate that a list is permitted as a list element only if all other elements are lists of the same type.

There are several important built-in list operators and functions, which we illustrate here. The built-in functions \text{hd} and \text{tl} correspond to \text{car} and \text{cdr} from Scheme. Therefore,

\[
\text{hd}([1,2,3]);
\]

\[
\text{val it = 1 : int}
\]

\[
\text{tl}([1,2,3]);
\]

\[
\text{val it = [2,3] : int list}
\]

\[
\text{hd}([1]);
\]

\[
\text{val it = 1 : int}
\]

\[
\text{tl}([1]);
\]

\[
\text{val it = [] : int list}
\]

\[
\text{hd}([]);
\]

\[
\text{uncaught exception Hd}
\]

\[
\text{tl}([]);
\]

\[
\text{uncaught exception Tl}
\]

The last two examples display the exception feature of ML, which we describe later.

ML employs the operator \text{:} to accomplish the job of Scheme's \text{cons} function—for example,
A second infix operator that applies to lists is the append operator, denoted by @ and illustrated by

```
[1,2,3] @ [4,5];
val it = [1,2,3,4,5] : int list
```

A prefix function rev is also defined to reverse a list.

```
rev[1,2,3];
val it = [3,2,1] : int list
```

It is convenient to define list functions via patterns, particularly using the :: operator to define a pattern on the list parameter. For example, if we wish to construct the recursive function for summing all the elements in a list, we could write it as

```
fun sum(nil) = 0
 | sum(first::rest) = first + sum(rest);
val sum = fn : int list -> int
```

Here the first pattern is the nil, or empty, list, which sums to zero and serves as the recursion stopper for this function definition. In the second pattern, first matches the head of the parameter list and rest matches the tail.

### 11.11.3 Parametric Polymorphism

ML does not require that parameters to functions be bound to a type when there is nothing in the function definition that requires such a binding. In this case, binding can be postponed until function application, when the type of the actual parameter can be used to infer types for each evaluation. This ability to define functions independently of the types of their parameters is called **parametric polymorphism**.

As a first example of this ability, consider the following function, which accepts a list as a parameter and returns the same list with the last element removed:

```
fun allbutlast(nil) = nil
 | allbutlast(x::nil) = nil
 | allbutlast(x::y) = x :: allbutlast(y);
val allbutlast = fn : 'a list -> 'a list
```

Notice the interesting new notation in the type description of allbutlast. This description indicates that the function accepts as its parameter a list of type 'a and returns a list of the same type. The notation 'a indicates that the type of the list is unrestricted, but the fact that it is used for both the parameter and the return value indicates that they must be lists of the same type. As an illustration, consider the following calls to allbutlast:

```
allbutlast([[1,2,3,4]]);
val it = [[1,2,3]] : int list
allbutlast(["a","b","c"]);
val it = ["a","b"] : string list
allbutlast([[1,2],[3,4],[5],[6,7]]);
val it = [[1,2],[3,4],[5]] : int list list
```
Such polymorphism is possible only when there is nothing in the function definition that
restricts the type. For example, the similar function that returns the smallest value in a list is not
legally defined.

```ml
fun smallest(nil) = nil
  | smallest(x::nil) = [x]
  | smallest(x::y) = if x<hd(smallest(y)) then [x]
                   else smallest(y);
Error: overloaded variable cannot be resolved: <
```

The problem here is the use of the operator `<`, which applies to three different types (int, real, string) and does not apply to other types. Therefore, the type of element in the para-
metric list is required to define which `<` is to be used in the definition of function `smallest`. A
definition that works for list of type int would be

```ml
fun smallest(nil):int list = nil
  | smallest(x::nil) = [x]
  | smallest(x::y) = if x<hd(smallest(y)) then [x]
                   else smallest(y);
val smallest = fn : int list -> int list
```

In this definition, the result of the function is bound to type int list, giving the `<` operator a
unique specification.

Let us examine one more function to illustrate parametric polymorphism. The function
`reversepair` accepts a pair of parameters and returns the same pair in the reverse order.

```ml
fun reversepair(x,y) = (y,x);
val reversepair = fn 'a * 'b -> 'b * 'a
```

Several new concepts are introduced by this function. First, it shows that functions in ML can
return multiple results. In this case, two results are returned as a tuple. Tuples are enclosed by
parentheses and are distinct from lists, which are enclosed by square brackets. Tuples have Carte-
sian product types. This function also illustrates that more than one unspecified type can occur in
the same function definition. Here the two parameters `x` and `y` are not bound to a type, so the vari-
able type specifications `'a` and `'b` are used. The fact that these are distinct names says that not
only are `x` and `y` not bound to a type in the function definition, but when they are bound during
an application of this function, they need not be bound to the same type. The following applica-
tions of this function illustrate this fact:

```ml
reversepair(1,"a");
val it = ("a",1) : string * int
reversepair([1,2],5.5);
val it = (5.5,[1,2]]) : real * int list
reversepair(1,5);
val it = (5,1) : int * int
```

### 11.11.4 Datatypes
ML permits the user to define enumerated types and discriminated union types, both
through the use of the keyword `datatype`. We first illustrate enumerated types.

```ml
datatype honor = none | CumLaude | MagnaCumLaude
```
Chapter 11 - July 26, 2002 Draft

| SummaCumLaude;
datatype honor
  con CumLaude : honor
  con MagnaCumLaude : honor
  con SummaCumLaude : honor
  con none = honor

This enumerated datatype may be used in any further function definition as well as in the definition of other datatypes. For example, a function that converts a grade point average (gpa) to an honor is given by

fun computehonor(gpa:real) = if gpa>=3.87 then SummaCumLaude
else if gpa>=3.6 then MagnaCumLaude
else if gpa>3.3 then CumLaude
else none
val computehonor = fn : real -> honor

The construction of **discriminated union** types is made possible through the same construct. A discriminated union type is a type that can take on any of several types. As an example, we construct a data type that is a discriminated union of int and int list. This is accomplished by

datatype listorint = alist of int list
  | anint of int;
datatype listorint
  con alist : int list -> listorint
  con anint : int -> listorint

Notice that functions named alist and anint have been created that convert from the types int list and int into listorint type. Therefore, these can be used to construct objects of this new type—for example,

alist([1,2]);
val it = alist [1,2] : listorint
anint(16);
val it = anint 16 : listorint

Functions can now be defined that detect the actual type of a parameter and perform the appropriate action. A function to sum objects of type listorint is constructed as follows:

fun sum(anint(x)) = x
  | sum(alist(nil)) = 0
  | sum(alist(x::y)) = x+sum(alist(y));
val sum = fn : listorint -> int

When this function is applied, its parameter must be of one of the acceptable types. Consider the results of the following applications:

sum(alist([1,2,3]));
val it = 6 : int
sum(anint(12));
val it = 12 : int
sum([1,2,3]);
In the last case, the error occurs because the int list has not been converted to type listint by converting using alist.

We close our discussion of the ML datatype definition with an illustration of how enumerated and discriminated types can be mixed and how the type variables can be used in the definition of types. We construct a representation of a binary tree over an unspecified type as follows:

```
datatype 'a binarytree = empty
        | node of 'a * ('a binarytree) * ('a binarytree);
datatype 'a binarytree
    con empty : 'a binarytree
    con node  : 'a * 'a binarytree * 'a binarytree
```

Here the binarytree datatype has two alternative definitions: empty, an enumerated value, and node, a type that is the Cartesian product of three types. The specification of 'a immediately after the keyword datatype indicates that 'a is a type parameter for this type definition.

Using the above definition for binarytree, a tree of any type can be constructed. For example, a simple tree of integers is constructed by

```
val tree = node(0,node(5,node(3,empty,empty),
            node(7,empty,empty)),
            node(15,node(12,empty,empty),
            empty)));
```

This represents the binary tree

```
    9
   / \
  5 15
 / \ / /
3  7 12
```

A function to search such a binary tree for a given value is

```
fun search(x,empty) = false
    | search(x,node(root,left,right)) = search(x,left) orelse
      x=root orelse
      search(x,right);
```

Note that this function performs the search via an inorder traversal. The binary tree tree constructed above, can be searched by applications of the function search such as

```
search(5,tree);
val it = true : bool
search(17,tree);
val it = false : bool
```

The general nature of the binarytree datatype is illustrated by the following sequence in ML, which creates a binary tree of strings and uses the same search function to search that tree.

```
val tree2 = node("nine",node("five",node("three",empty,empty),
```

```
search(tree2,"nine");
val it = true : bool
search(tree2,"four");
val it = false : bool

11.11.5 Exceptions

Another very useful feature of ML is the inclusion of exceptions that can be raised at any point in a function application. For example, when we constructed the function allbutlast, we returned an empty list when an empty list was the parameter, even though the function should be undefined in that case. A more appropriate action might be to raise an exception when that situation occurs. The revised version of the function is

```
exception EmptyParameter;
val EmptyParameter : exception
fun allbutlast(nil) = raise EmptyParameter
    | allbutlast(x::nil) = nil
    | allbutlast(x::y) = x::allbutlast(y);
val allbutlast = fn : 'a list -> 'a list
```

Then, calling this function with an empty parameter gives

```
allbutlast(nil);
uncaught exception EmptyParameter
```

Exceptions can be handled in the environment where a function raising the exception is called--for example,

```
fun tryabl(aList) = (allbutlast(alist),"good")
    handle EmptyParameter => (nil,"no good");
val tryabl = fn : 'a list -> 'a list * string
```

```
tryabl([1,2,3]);
val it = ([1,2],"good") : int list * string
tryabl([]);
val it = ([],"no good") : 'a list * string
tryabl([1.0]);
val it = ([],"good") : real list * string
tryabl(allbutlast(['a']));
val it = ([],"no good") : string list * string
```

Unhandled exceptions are propagated up to the calling environment, meaning if a function does not handle a raised exception, the same exception is raised in the function calling the first function.

11.11.6 First Class Functions

ML treats functions as first class objects, permitting them to be used as parameters and return values in function definitions. This powerful feature is implemented in a very natural way, as illustrated by the following function:
fun twice(f) = fn x => f(f(x));
val twice = fn : ('a -> 'a) -> 'a -> 'a

The function twice returns a function that is the application of the parametric function \( f \) to its parameter twice. For example, this function can be applied to factorial as follows:

val twinfact = twice(factorial);
val twinfact = fn : int -> int
val it = 720 : int

twinfact(3);
val it = 720 : int

Note that the function twice is polymorphic in that it applies to functions on any type. It only requires that the function’s domain and range types be the same. Applying twice to the list function \( \text{tl} \) gives

val twintail = twice(tl);
val twintail = fn : 'a list -> 'a list
val it = [3,4] : int list
val it = ["c"] : string list

This function can even be applied to itself.

val twintwice = twice(twice);
val twintwice = fn : ('a -> 'a) -> 'a -> 'a
val quadtl = twintwice(tl);
val quadtl = fn : 'a list -> 'a list
val it = [] : int list

Further examples of higher-order functions are found later when common functional forms are implemented in ML.

11.12 Examples

In this section we develop ML functions that correspond to the example programs presented earlier in FP and Scheme.

11.12.1 Example 1 - Factorial

Although a factorial function was presented earlier in this chapter, we include another version here that raises an exception when the parameter is negative.

exception NegativeParameter;
exception NegativeParameter
fun factorial(n) = if n< 0 then raise NegativeParameter
  else if n=0 then 1
  else n*factorial(n-1);
val factorial = fn : int -> int

11.12.2 Example 2 - Quicksort
The approach used to construct quicksort in Scheme can be duplicated in ML. We construct the functional form `keep_some` that keeps only those elements of a list for which an application of `f` results in a true value.

```
fun keep_some(f) = fn(x,alist) => if alist=nil then nil
  else if f(hd(alist),x) then
    hd(alist)::(keep_some(f))(x,tl(alist))
  else (keep_some(f))(x,tl(alist));
val keep_some = fn : ('a * 'b -> bool) -> 'b * 'a list -> 'a list
```

Next we construct two functions to which `keep_some` can be applied. At this point we restrict ourselves to type int.

```
fun gtr(x,y:int) = x>y;
val gtr = fn : int * int -> bool
fun leq(x,y:int) = x<=y;
val leq = fn : int * int -> bool
```

We can now define two versions of `keep_some`, one for `gtr` and one for `leq`.

```
val keep_gt = keep_some(gtr);
val keep_gt = fn : int * int list -> int list
val keep_le = keep_some(leq);
val keep_le = fn : int * int list -> int list
```

Quicksort can now be constructed from these two functions.

```
fun quicksort(nil)=nil
  | quicksort(x::y)=quicksort(keep_le(x,y)) @ [x] @ quicksort(keep_gt(x,y));
val quicksort = fn : int list -> int list
```

In order to modify quicksort to apply to different types, new definitions of `gtr` and `leq` must be constructed and `keep_gt`, `keep_le` and `quicksort` must be re-instantiated, using the same definitions as before.

```
fun gtr(x,y:string) = x>y;
val gtr = fn : string * string -> bool
fun leq(x,y:string) = x<=y;
val gtr = fn : string * string -> bool
val keep_le = keep_some(leq);
val keep_le = fn : string * string list -> string list
val keep_gt = keep_some(gtr);
val keep_gt = fn : string * string list -> string list
fun quicksort(nil) = nil
  | quicksort(x::y) = quicksort(keep_le(x,y))@ [x] @ quicksort(keep_gt(x,y));
val quicksort = fn : string list -> string list
```
ML also contains more advanced data abstraction features, which permit a better generic definition of functions such as quicksort, but those are beyond the scope of the present discussion.

11.12.3 Example 3 - Stacks

We again use lists as the representation for stacks, with the first element of the list representing the top element. Our three stack operations are then defined as follows:

```ml
fun empty(nil) = true
    | empty(x) = false;
val empty = fn : 'a list -> bool
fun push(stack,element) = element::stack;
val push = fn : 'a list * 'a -> 'a list
exception PopEmpty;
exception PopEmpty
fun pop(nil) = raise PopEmpty
    | pop(x::y) = (x,y);
val pop = fn : 'a list -> 'a * 'a list
```

11.12.4 Example 4 - Attribute Lists

We implement attribute lists in ML as a list of 2-tuples. This approach is used rather than the list of lists used in Scheme because a list of lists in ML would require the attribute name and the attribute value to be of the same type. This requirement exists because of the homogeneous nature of ML lists. To facilitate this approach, we need an auxiliary function that extracts the first element from a 2-tuple.

```ml
fun first(x,y) = x;
val first = fn : 'a * 'b -> 'a
```

Next we construct the three attribute list functions.

```ml
fun addattr(attlist,aname,avalue) = (aname,avalue)::attlist;
val addattr = fn:('a * 'b) list * 'a * 'b->('a * 'b) list
fun rmattr(nil,aname) = nil
    | rmattr(x::y,aname) = if first(x)=aname then
                         rmatrr(y,aname)
                         else x::rmattr(y,aname);
val rmattr = fn : ('a * 'b) list * 'a -> ('a * 'b) list
fun getattr(nil,aname) = nil
    | getattr(x::y,aname) = if first(x)=aname then
                             x::getattr(y,aname)
                            else getattr(y,aname);
val getattr = fn: ('a * 'b) list * 'a -> ('a * 'b) list
```

11.13 Comparison of ML to the Functional Model

11.13.1 Departures from Functional Model

ML is similar to Scheme in its departures from the functional model. The use of variables that create a side effect in the sense of modifying some permanent store, is implemented by means of references in ML. References correspond to pointer variables.

Furthermore, sequential execution is also possible in ML. This is illustrated by the following function, which prints a count down from its parametric value to 1.
fun count_down(0) = print("\n")
| count_down(n) = (print(n);
| print(" ");
| count_down(n-1));
val count_down = fn : int -> unit

10 9 8 7 6 5 4 3 2 1
val it = () : unit

The built-in print function is used to produce printed results. Functions such as print, which have no result but exist for their side effects, are said to return type unit. In the case of the definition of count_down, three statements are collected into a sequential block, and they are enclosed in parentheses. This is a violation of the functional model requirements.

11.13.2 Functional Forms in ML

Although ML has some of the common functional forms built in as operators or functions, we derive definitions for some of them in this section to illustrate the way ML treats functions as first-class objects.

We begin with composition, which is directly defined by

fun composition(f,g) = fn x => f(g(x));
val composition = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

Note how the types are inferred in this definition. This function can be applied directly to any pair of functions where the domain of the first matches the range of the second. Some examples are

composition(factorial,factorial)(3);
val it = 720 : int
composition(hd,tl)([1,2,3]);
val it = 2 : int

This functional form is also built in to ML in the operator o. Therefore, the preceding action application could also be written

(hd o tl)([1,2,3]);
val it = 2 : int

The constant functional form which produces a function that always returns the same constant value. It is written in ML as

fun constant(n) = fn y => x;
val constant = fn : 'a -> 'b -> 'a

The construction functional form applies a list of functions to a single common parameter, producing a list of results. This is expressed in ML by

fun construct(flist) fn x => if length(flist)=0 then nil 
else (hd(flist))(x)::construct(tl(flist))(x);
val construct = fn : ('a -> 'b) list -> 'a -> 'b list

This function can be applied as follows:
(construct([sqrt, exp, ln, sin])(2.0);
[1.414213, 7.389056, 0.693147, 0.909297] : real list

The apply_to_all functional form applies a single function to a list of domain values, producing a list of resulting range values. This is provided in ML as the built-in function map, but can also be constructed as follows:

\[
\text{fun apply\_to\_all(f) = fn x => if x=nil then nil else f(hd(x))::apply\_to\_all(tl(x));}
\]

\[
\text{val apply\_to\_all = fn : ('a -> 'b) -> 'a list -> 'b list}
\]

The insertion functional form takes as its parameter a two-parameter function, the first parameter being a function and the second parameter being a list of elements form the domain of the first parameter. When applied to a list of this type, insertion constructs a result that cumulatively applies the parametric function to elements from the list, with applications going from right-to-left. The ML version of the insertion functional form is

\[
\text{fun insertion(f) = fn x => if length(x)=1 then hd(x) else f(hd(x),insertion(f)(tl(x));}
\]

\[
\text{val insertion = fn : ('a * 'a list -> 'a list) -> 'a list -> 'a list}
\]

If we wish to apply this to an arithmetic operator such as + or -, we first need to convert the operator from infix to prefix (called nonfix by ML) form. This is done by writing

```
nonfix +;
nonfix +
3+4;
Error: operator is not a function
+(3,4);
val it = 7 : int
```

We can now apply + using insertion as seen below.

```
insertion(+)([1,2,3,4,5]);
val it = 15 : int
nonfix -;
nonfix -
insertion(-)([1,2,3,4]);
val it = -2 : int
```

The final application illustrates that insertion applies its function from right-to-left.

11.14 Evaluation of functional programming languages

When compared to imperative languages, functional languages offer several advantages. Because the structure of the program is related to the computation rather than the machine, it is easier to construct programs for problems that are functional in nature. Contributing to the efficiency of learning and using functional languages is the fact that their syntactic structures are much simpler than those of imperative languages.

The referential transparency of functional languages means that the order of function evaluations is irrelevant to the computation. This permits functional languages to be easily used for concurrent processing since function applica-
tions may be applied in any order without concern for dependencies in the computation. The referential transparency also makes it easier to reason about the correctness of a program since the correctness of any function is unaffected by side effects from other computations.

There are also disadvantages to functional languages. The functional approach means that efficient execution of programs is more difficult since the structure of the program differs significantly from the structure of the machine on which it is run. Functional languages also require more extensive notation which either results in long, complex expressions as in Lisp, or a cumbersome alphabet as in APL or FP.

Perhaps the most significant drawback to functional languages is the fact that many programmers find it difficult to think in the functional paradigm. It is believed by advocates of functional languages, that this is a consequence of programmers first learning programming with an imperative language. Many such as Abelson et al (1985) have designed introductory courses around the functional paradigm to address this issue.

Terms - Chapter 11

function
domain
range
lambda expression
bound variables
function composition
functional form
referential transparency
first class objects
lazy function evaluation
functional form
atom
list
cell model
car
cdr
S-expression
predicate
ML
meta language
parametric polymorphism
discriminated union

Discussion Questions - Chapter 11

1. What do you see as advantages of programming in the functional paradigm? What are disadvantages?

2. What extensions would you add to the functional model to make it easier to use? How would these affect the integrity of the model?

3. Discuss the types of problems that are best suited for functional programming solution.

4. As discussed in this chapter, Scheme goes beyond the functional model in several ways. What are the advantages of doing this? What are the disadvantages?

5. Think about data structuring in Scheme. In considering Discussion Question 4, several facilities in Scheme that go beyond the functional model were discussed. Are records or arrays possible in Scheme? How would you implement something like a tree in Scheme?
6. Would pointers be useful in Scheme? Why or why not?

7. Both functions and data are expressed in the form of an S-expression in Scheme. Why might this be useful?

8. How does the concept of abstract data type apply to Scheme?

9. LISP dialects like Scheme are preferred by many computer scientists. For example, artificial intelligence researchers prefer LISP for applications like planning systems and game playing strategy developers. Discuss the features of LISP that make it desirable for these applications.

10. What are some advantages to ML’s strict typing over the more relaxed view of types taken by Scheme? What are some disadvantages?

11. Functions in ML can be defined using the \texttt{fun} keyword or the \texttt{val} keyword. Which do you think is better and why?

12. ML lists must be homogeneous, whereas Scheme’s lists may be heterogeneous. What are some advantages to each of these strategies?

13. Compare the exception handling feature of ML with that of Java.

Exercises - Chapter 11

1. Write lambda expressions for the following functions typical found on calculators:
   \[ x^2 \]
   \[ \text{square root} \]
   \[ \text{the slope of a line} \]
   \[ \text{pound-to-kilogram conversion} \]
   \[ \text{Fahrenheit to Celsius conversion} \]

2. For each of the functions expressed in Exercise 1, show a demonstration application.

3. Define a lambda expression to compute \( ax^2 + by + z \). Show an application for \( a = 2, \ b = 3, \ x = 4, \ y = 6, \ z = 9 \).

4. Consider the depiction of a LISP S-expression given in Figure 11.7. If the structure is the value \( \text{“lst,”} \) what is the value of \( \texttt{(cdaadr lst)} \)

5. Write the Scheme code necessary to add the following structure to \( \text{“d,”} \) as in Figure 11.8.

6. Using the cell model of Scheme, depict (that is, draw) the list \( \texttt{((b c) (x (y z)))} \)

7. Using only the functions \texttt{car}, \texttt{cdr}, \texttt{cond}, and itself (that is, recursively), write a Scheme function called \texttt{rev} that takes a list and returns a list that is a reversed version of the original.

8. Using the cell model for Scheme, depict the internal representation of the following lists:

   (a) \( \texttt{(a (b c) d e (f))} \)
   (b) \( \texttt{a b (c (d e) f)} \)
   (c) \( \texttt{(a b (c) d (e f))} \)
9. Write a set of Scheme functions to implement a binary search tree. Assume that the value stored in each node is an integer and a tree is represented by a three-element list. The first element of the list is the value at the root, the second element is the left subtree of the root, and the third element is the right subtree of the root. The subtrees are represented by similarly formed lists. If there is no subtree, its place is filled by the empty list. You should implement three functions 1) create, which takes no parameters and returns an empty tree; 2) insert, which takes as parameters a tree and an integer and returns the tree with the integer correctly inserted; and 3) search, which takes as parameters a tree and an integer and returns #t or #f depending on whether the value was present in the tree.

10. We can represent a 3 x 3 matrix in Scheme by using a list of lists. For example, the matrix

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{array}
\]

can be represented by the list `((1 2 3) (4 5 6) (7 8 9)).

a. Write a Scheme function to sum the items in a column of such a 3 x 3 matrix. Call the function by `(column matrix 2)` where `matrix` is the matrix as a list and 2 is the column to sum.

b. Write a Scheme function that sums all numbers in a matrix. Call this by `(summation matrix)` where `matrix` is the list representing the matrix whose elements are to be summed.

11. What is the type of each of the following ML expressions? If the expression is illegal because of type, explain why.

a. `6+2;`

b. `6/2;`

c. `6 div 2;`

d. `fun d(x,y) = x+y;`

e. `fun e(x,y) = x+1;`

f. `fun f(x) = x::x;`

g. `fun g(x) = x::[x];`

12. The function `repeat` was defined in this chapter using the `if-then-else` construct of ML. Create a pattern matching definition for the same function.

13. The following functions were defined in this chapter using the pattern matching construct of ML. Create `if-then-else` definitions for each of them.

```
sum(x:int) sum(x:listorint)
allbutlast search
smallest empty
pop
```

14. Construct a function that applies to two parameters of type `listorint` in the following way:

```
add(x,y) = x+y if x and y are both integers
add(x,y) = [x+y_1 ... x+y_n] if x is an integer and y=[y_1 ...y_n]
add(x,y) = [x_1+y ...x_n+y] if x=[x_1 ...x_n] and y is an integer
add(x,y) = [x_1+y_1 ...x_n+y_n] if x and y are both lists
```
15. Construct the following functional form in ML:

\[
\text{prod}(f,g):x = (f(x), g(x))
\]

\[
\text{list_to_list}([f_1 \ldots f_n]):[x_1 \ldots x_n] = [f_1(x_1) \ldots f_n(x_n)]
\]

16. Write an ML function that accepts as its parameter a list of integers and returns as its result the sum of the squares of those integers.

17. Recall the Chapter 12 exercise where we represented a 3 x 3 matrix as a Scheme list. Do the same for ML in constructing the following two functions:

a. Write an ML function to sum the items in a column. Compare this with the Scheme function that you wrote to accomplish the same task.

b. Write an ML function that sums all the numbers in a matrix. Compare this to the Scheme function you wrote to accomplish the same task. Specifically, compare how you used functional forms in ML versus the method you used to combine functions in Scheme.

Laboratory Exercises - Chapter 11

1. This exercise involves the implementation of a heavily recursive program that implements a solution in Scheme to the \textit{n-queens problem}.

The \textit{n-queens problem} is a classic combinatorial problem where the object is to place \(n\) queens on an \(n \times n\) chessboard so that no two can "attack" each other, that is, so that no two of them are on the same row, column, or diagonal. An iterative algorithm that computes a solution to the \textit{n-queens problem} is given below:

\[
\text{algorithm NQUEENS}(n);
\]

\[
\text{var } n, c, r: \text{integer}; \text{ board: boardtype} ;
\]

\[
\text{begin}
\]

\[
\text{board}:=\text{newboard};
\]

\[
c:=0; \text{ r}=1;
\]

\[
\text{while } r>0 \text{ do}
\]

\[
c:=c+1;
\]

\[
\text{mark(board, c, r)};
\]

\[
\text{while } c_n \text{ and not goodplace(board, c, r) do}
\]

\[
\text{unmark(board, c, r)};
\]

\[
c:=c+1;
\]

\[
\text{mark(board, c, r)};
\]

\[
\text{endwhile};
\]

\[
\text{if } c_n \text{ then}
\]

\[
\text{if } r=n \text{ then}
\]

\[
\text{print("Solution found");}
\]

\[
\text{exit};
\]

\[
\text{else}
\]

\[
r:=r+1;
\]

\[
c:=0;
\]

\[
\text{endif};
\]

\[
\text{else}
\]

\[
r:=r-1;
\]

\[
c:=\text{place(board, r)};
\]

\[
\text{unmark(board, c, r)};
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{algorithm NQUEENS}(n);
\]

\[
\text{var } n, c, r: \text{integer}; \text{ board: boardtype} ;
\]

\[
\text{begin}
\]

\[
\text{board}:=\text{newboard};
\]

\[
c:=0; \text{ r}=1;
\]

\[
\text{while } r>0 \text{ do}
\]

\[
c:=c+1;
\]

\[
\text{mark(board, c, r)};
\]

\[
\text{while } c_n \text{ and not goodplace(board, c, r) do}
\]

\[
\text{unmark(board, c, r)};
\]

\[
c:=c+1;
\]

\[
\text{mark(board, c, r)};
\]

\[
\text{endwhile};
\]

\[
\text{if } c_n \text{ then}
\]

\[
\text{if } r=n \text{ then}
\]

\[
\text{print("Solution found");}
\]

\[
\text{exit};
\]

\[
\text{else}
\]

\[
r:=r+1;
\]

\[
c:=0;
\]

\[
\text{endif};
\]

\[
\text{else}
\]

\[
r:=r-1;
\]

\[
c:=\text{place(board, r)};
\]

\[
\text{unmark(board, c, r)};
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{end}
\]
endif;
endwhile;
endalgorithm

The preceding algorithm requires a few notes. First, note that board is defined in terms of boardtype, which is not specified. Note as well that board is manipulated by three functions: newboard, which initializes the board representation to an empty board, mark, which places a queen on board at column c and row r, and unmark, which removes the queen at column c and row r on board. Finally, note that the function goodplace is undefined. This function determines if a queen placed on column c and row r on board can be attacked. You must also provide implementation for a function called place which takes a board and a row as arguments, and returns the column in which the row’s queen is placed.

You are to implement a recursive solution to the n-queens problem in Scheme. As a guide, follow the steps below:

1. Rewrite the preceding algorithm so that it is defined recursively and is phrased as functionally as possible.
2. Design a data structure (using Scheme lists and atoms) to represent the board. Then write the supporting routines newboard, mark, and unmark.
3. Write the function goodplace in terms of the data structure designed above.
4. Finally, write the function NQUEENS. This function should take one argument, the number of queens in the problem, and produce some indication of where the queens are placed. For example, you might print an entire board, or simply an indication of row and column placement of queens.

You should also keep these points in mind:

1. For a given n, there are many solutions to the n-queens problem. Your program need find only one.
2. You may alter the number and type of arguments used in the functions in the description of the preceding NQUEENS algorithm. Of course, you may introduce your own functions.
3. The function NQUEENS must take only one argument, however.

2. Verify your answer to Exercise 9 by implementing the BST specification in Scheme. Example output from this type of implementation is shown below:

```
1 ]=> (define tree1 (insert '() 50))
;Value: tree1
1 ]=> tree1
;Value: ((() 20 ()) 50 (() 70 ()))
```

3. Construct a set of string functions in ML as specified:

```
charinstring(n,s) returns the nth character in string s
position(s1,s2) returns the integer position of the leftmost
 occurrence of s1 in s2, empty if not found
count(s1,s2) returns a count of the number of times s1
occurs in s2
wordcount(s) returns a count of the number of words in s
 where a word is defined as any sequence of
 letters delimited by non-letters, beginning
 of string, or end of string
```
4. Construct a set of functions in ML that simulates complex arithmetic by accepting as parameters and returning as values pairs of reals to represent complex numbers.

   realpart(c) returns the real part of complex number c
   imagpart(c) returns the imaginary part of c
   cadd(c1,c2) returns c1+c2
   csub(c1,c2) returns c1-c2
   cmult(c1,c2) returns c1*c2
   cdiv(c1,c2) returns c1/c2
   cabs(c) returns the real number that is the modulus of c

5. Construct a discriminated union data type that can be either real or int and define add(x,y) over this type to perform the correct addition no matter what the base types of the two parameters and to return a real result.

6. Construct a set of functions to perform binary search tree functions on the binarytree type defined in this chapter. Functions should include

   insert(x,bst) delete(x,bst) printinorder(x,bst)

7. Consider the NQUEENS problem from Chapter 12. Implement this algorithm in ML. Compare the resulting code with that developed in Scheme. What advantage did you derive from the function-oriented nature of the two languages. What advantage did you derive from the strong typing of ML.
Figure 11.7

Figure 11.8