Part III

Application Modeling

Chapter 12

Functional Languages

Overview

This chapter presents the goals and attractions of functional programming. Functional languages have developed as semantically clean ways to denote objects, functions, and function application.

The concept of denotation, as opposed to computation, is introduced. Functional constructs denote objects, in contrast to procedural constructs which specify methods for constructing objects.

Development of functional languages started with lambda calculus, which was extended to denote types, literals, and operations for primitive data domains. Lazy evaluation and the absence of destructive assignment were also carried over from lambda calculus into the modern functional languages.

Functional languages have a minimum of explicit control structures. Explicit loops are replaced by tail recursion and implicit iterative constructs. Sequences of statements can be replaced by lazy evaluation, nested context blocks, or nested procedure calls because there is no destructive assignment.

Miranda is presented as an example of a functional language because of its generality, power, and simple, attractive syntax. List comprehensions and infinite lists are explained, and applications are given.

12.1 Denotation versus Computation

During the 1960s, several researchers began work on proving things about programs. Efforts were made to prove that:

- A program was correct.
- Two programs with different code computed the same answers when given the same inputs.
- One program was faster than another.
- A given program would always terminate.

While these are abstract goals, they are all, really, the same as the practical goal of "getting the program debugged".

Several difficult problems emerged from this work. One was the problem of *specification*: before one can prove that a program is correct, one must specify the meaning of "correct", formally and unambiguously. Formal systems for specifying the meaning of a program were developed, and they looked suspiciously like programming languages.

Researchers began to analyze why it is often harder to prove things about programs written in traditional languages than it is to prove theorems about mathematics. Two aspects of traditional languages emerged as sources of trouble because they are very difficult to model in a mathematical system: mutability and sequencing.

Mutability. Mathematics is a pure denotational system; a symbol, once defined in a given context, means one thing. The truth or falsity of a mathematical statement is a permanent property of the statement; it does not change over time. But programming languages have variables, whose contents can be changed at any time by a remote part of the program. Thus a mathematical proof about a program that uses assignment is necessarily nonmodular and must consider all parts of the program at once.

Programming language functions are not like mathematical functions because of the mutability of variables. In mathematics, a function is a mapping from one set of values (the domain) onto another (the range). To use a function you supply an `argument` from the domain, and the function returns an element from the range. If the function is used twice with the same argument, the element returned will be the same. We can thus say that a mathematical function f denotes a set of ordered pairs $(x, f(x))$.

It is easy to write a program that does not correspond to any mathematical function. Exhibit 12.1 shows a definition of a `C` function that, if called twice with the same argument, returns different answers. This function increments a static local variable each time it is called, and returns the sum of the argument and the value of that variable. Although this code uses a static variable, which is not supported by many languages, the same function could be written in almost any language by using a global variable instead.

Exhibit 12.1. A nonfunctional function in C.

```
int my_func(int x)
{   static int z = 0;      /* Load-time initialization.  */
    return x+(z++);        /* Value of z increases after each call.  */
}
```

Sequencing. When a mathematician develops a theorem, she or he defines symbols, then writes down facts that relate those symbols. The order of those facts is unimportant, so long as all the symbols used are defined, and it certainly does not matter where each fact is written on the paper! A proof is a static thing—its parts just *are*, they do not *act*.

In contrast, a program is a description of a dynamic process. It prescribes a sequence of actions to be taken, not a collection of facts. Sometimes the order of a pair of program statements does not matter, but other times it does. Some statements are in the scope of a loop and are evaluated many times, each time producing a different result. To fully describe a program during execution (which we call a *process*), we need not only the program code, but a copy of the stack, the values of global and static variables, the current values of the computer's program counter and other hardware registers, and a copy of the input.

12.1.1 Denotation

Functional languages are an attempt to move away from all this complexity and toward a more mathematical way of specifying computation. In a functional language, each expression of the language *denotes* an object. Objects are pure values and may be primitive data elements such as integers, functions, or higher-order functions. Thus each expression can be thought of as a description of a static, mathematical entity rather than of a dynamic computation.

Some examples will help make this clear. The expression 3 denotes the integer 3, as do the expressions $2+1$ and $5-2$. The meanings of these expressions are fixed for all time and do not change depending on what has happened elsewhere in the program. In functional languages, these same properties are extended to functions. (Remember that a function is a set of ordered pairs.) The factorial function, `fact(n)`, takes an integer argument n and returns a number $n!$, which is the product of the first n natural numbers. This mathematical function might be described in a functional way by a pair of *recurrence equations*, for example,

$$\begin{aligned} \text{fact } 0 &= 1 \\ \text{fact } n &= n * \text{ fact } (n-1) \text{ if } n \geq 1 \end{aligned}$$

The first equation describes the value of `fact` for argument 0. The second equation describes how the value of `fact n` is related to the value of `fact(n - 1)`. It take some effort to show that there is a unique function (on the natural numbers) that satisfies these equations and hence that

the equations do in fact specify a function, but it is clear from the form of the equations that whatever the meaning is, it does not depend on the history of execution up to this point. We say that this pair of equations denote the mathematical function $n!$.

Denotational semantics is a way of describing the meaning of a programming language construct by giving the mathematical object that it denotes. In the above examples, the meaning of `2+1` is the integer 3, and the meaning of the two equations specifying `fact` is the factorial function. Contrast this style of description with the Pascal program for computing factorial shown in Exhibit 11.16.

What this Pascal expression denotes is the process of successively multiplying `temp` by the integers $1, 2, 3, \ldots, n$. In this case, the result of that process happens to be functional, and the program does indeed compute the factorial function, but it is not obvious without careful inspection of the code that the program in fact computes a function.

Functions are denoted by λ-expressions. For example, consider the function G denoted by the expression

$$\text{Let } G \text{ be } \lambda f.\lambda x.(\text{ if } x = 0 \text{ then } 1 \text{ else } x * f(x-1)).$$

G denotes a function of two arguments, f, a function, and x, an integer. Suppose p is the function $p(n) = 3n$. Then $Gp5$ denotes the value of the expression

$$\text{if } 5 = 0 \text{ then } 1 \text{ else } 5 * p(5-1)$$

which is equal to $5 * p(4) = 5 * 12 = 60$.

Fixed Points. The function G above has a curious property: namely, if h is the mathematical function factorial(n), then $G(h)$ is also the factorial function. We say that the factorial function is a *fixed point* of G because $h = G(h)$. Moreover, it is the least fixed point in the sense that any function h' satisfying the equation $h' = G(h')$ agrees with factorial at all of the points where factorial is defined (the nonnegative integers). We can use this fact and the "least fixed point operator", Y, to write an expression that denotes the factorial function, namely YG, or equivalently,

$$Y\lambda f.\lambda x.(\text{ if } x = 0 \text{ then } 1 \text{ else } x * f(x-1))$$

Intuitively, $Y\lambda f$ just means, "in the following expression, f refers recursively to the function being defined".

A curiousity of lambda calculus is that Y itself can be denoted by a (rather complicated) lambda expression. Hence Y does not have to be added to the language as a special primitive for defining recursive functions. In practice, it almost always is added for efficiency reasons. Rather than replace Y by the equivalent lambda expression, it is much more efficient in the implementation to allow pointers to functions; then $Y\lambda f \ldots$ simply binds f to the function being defined by the $\lambda x \ldots$ expression. Lambda calculus is a sufficient semantic basis for expressing all computations *because* it is able to express the Y operator as a formula.

The fact that pure lambda calculus lacks types is not an oversight; it is necessary in order to define Y. The reason is that Y works with *any* function as its first argument, not just with functions

of a specific type.[1] Imposing a type structure on lambda calculus, as many functional languages do, weakens the lambda calculus and makes it necessary to introduce additional constructs to allow for recursive definitions. Convenience of use also dictates the addition of other constructs. Still, it is elegant that so simple a system as the untyped lambda calculus could be the basis of a powerful programming language!

12.2 The Functional Approach

Abstractly a functional language is simply a system of notation for denoting primitive values and functions. Lambda calculus, which was already proven to be a fully general formal system in which any computable function could be defined, was used as a notation for early functional languages. However, lambda calculus has two serious shortcomings as a language for programmers: it lacks any concept of types and it is hopelessly clumsy to use.

Nevertheless, lambda calculus has been taken as the starting point for the development of most real functional languages. The earliest practical language in this family was LISP, which now has two important descendants, Common LISP and Scheme.

Another class of languages has been developed recently, which we will call *pure functional languages*. These languages follow lambda calculus and functional principles much more closely. This family includes ML, Miranda, and Haskell. An extended version of lambda calculus was developed which includes types and operators and literals for the underlying data domains. This system forms the semantic basis upon which the pure functional languages are built. Functional programs are parsed and translated into computation trees which represent lambda applications. At execution time, the lambda evaluator crawls over these trees, binding arguments, evaluating expressions as needed, and replacing expression trees by constant values wherever possible.

Pure functional languages come extremely close to implementing the semantics that Church defined for lambda calculus. They share many traits with the older functional languages, specificly, functions are first-class objects, conditional expressions are used, rather than conditional statements, and repetition is done implicitly or with recursion, not by explicit loops. The key differences between the newer and the older functional languages are:

- The prohibition on destructive assignment.
- The complete absence of statement sequences.
- The use of lazy parameter evaluation.

We will consider each of these issues and show how denotational constructs can take the place of procedural language elements.

[1]The type of a function is defined by the types of its argument and return value.

12.2.1 Eliminating Assignment

Destructive assignment causes semantic problems because it creates a time- and sequence-dependent change in the meaning of a name. Prohibiting destructive assignment does not imply that a programmer cannot give a name to a value, merely that a name, once bound to a value, must remain bound to the same value throughout the scope of the name. Thus the meaning of a name is constant, and not dependent on the order of execution. This greatly simplifies the task of understanding the program and proving that it is correct.

In procedural languages, assignment is used for several purposes:

1. To store the result of computing a subexpression.
2. In conjunction with conditional statements.
3. With a loop variable, to mark the progress of a loop.
4. To build complex data structures.

To show that assignment can be eliminated, we must show how each of these things can be accomplished, with reasonable efficiency and style, in a functional way.

Programmers store the results of subexpressions goal (1) for many good reasons. When an expression is complex, a local name can be used to "break out" and name a portion of it, so that the structure and meaning of the entire expression can be made clearer. Local names are also used in Pascal for efficiency; rather than write a common subexpression several times, the subexpression is evaluated first, and its result is assigned to a local variable. The local name is then used in further computations. Both uses of local variables can reduce the complexity of expressions and make a Pascal program easier to understand and debug. It is easy to devise a functional method to meet this need. We simply need to let the programmer open a local context block in which to define local names for expressions. Miranda provides three ways to define a symbol:

- As a global symbol, by declaring the name with an "`=`", just as symbols are defined in lambda calculus.
- As a parameter name. The name is bound to an argument expression during a function call.
- As a local symbol, by using a `where` clause.

A Miranda `where` clause defines one or more local names and gives an expression for the meaning of each. The entire group of definitions forms a local context and can be implemented by a stack frame. `Where` clauses can also be nested, producing block structured name-binding.

Goal (2) is satisfied by using a conditional expression, which returns a value, instead of a conditional statement. The value returned by the conditional expression can be bound to a name by parameter binding or by a `where` clause.

Extensive support for list data structures, including list constructors, whole-list operations with implied iteration, and expressions that denote complex lists all combine to minimize the need for loops, goal (3). Sequences of data values are represented as lists, and operations that process such

Exhibit 12.2. Primitive operations for arrays in APL.

APL (*Array Processing Language*) was designed to make working with arrays easy. It contains more kinds of array primitives than other languages, including operations such as generalized cross-section, concatenation, lamination, and whole-array arithmetic. It gives excellent support for all sorts of array manipulation.

In these examples, the variables named `ar` and `br` are 3 by 3 arrays.

Array Primitive	Example in APL
Exhibit a literal array	17 18 19
Fetch dimensions of array	ρ ar
Create and initialize an array	ar ← 3 3 ρ 0
Subscript (random access)	ar[1;3]
Cross section	ar[1 2; 2]
Change one array item	ar[1,1] ← 17
Prepare for sequential access	ix ← 1
Access current item	ar[ix; iy]
Whole-matrix operations	ar + 1
Concatenation on dimension n	ar ,[1] br
Lamination (makes more dimensions)	ar ,[.5] br

sequences are written by mapping simple operations over lists. This device eliminates the need for most explicit loops and many statement sequences. The remaining repetitive processes are programmed using recursion. These topics are discussed at length in Sections 12.2.2 and 12.3.

Goal (4) is the only real difficulty. Prohibiting destructive assignment means that a data structure cannot be modified after it is created. For small data structures this is no problem, as a new copy of the structure can be created that incorporates any desired change. This, in fact, is how call-by-value works: objects are copied during the function call, and the new copies are bound to the parameter names within the function. Most list processing and a lot of processing on records and arrays can be done efficiently this way. The exception is when a large data structure, such as a 1000 element array, is used as a parameter and slightly modified within the subroutine. Copying it all in order to modify one element might produce unacceptable inefficiency.

Efficient implementation of arrays in a functional language is a current research topic. Work centers on the question of whether the *original* and *modified* versions of the array are both used. If not, then the language system can perhaps avoid the copy operation. It can *do* a destructive assignment to an array element "on the sly" and pretend that it didn't. This trick will succeed if the program never again asks to see the original version of the array. Because of the efficiency problem with large arrays, existing pure functional languages are all list-oriented.

When (and if) ways are found to circumvent these problems, careful design efforts will be needed to integrate arrays into functional languages. APL is an array-processing language that

has many things in common with functional languages and lets the programmer use a functional programming style to do complex combinations of whole-array operations. To support this style, APL has a large and complex set of array operations [Exhibit 12.2]. The set of primitive composition and decomposition operations needed to make arrays usable without constant use of assignment is large compared to the set provided for list handling in Miranda [Exhibit 12.11]. Today's list-oriented functional languages are small and simple; tomorrow's languages, with arrays added, will be larger and more complex.

12.2.2 Recursion Can Replace `WHILE`

In Chapter 10, we discussed the fact that all programs can be written in terms of three control structures: statement sequence, `IF...THEN...ELSE`, and the `WHILE` loop. That is, these three control structures are *sufficient* for all programming needs. However, they are not *necessary*; both `WHILE` and sequence can be eliminated. An alternative set of *necessary control structures* is:

- `IF...THEN...ELSE`
- recursive procedure call

Repetition of sections of code is certainly fundamental to programming; problems that do not involve repetition are usually solved by hand, not by computer. But repetition does not have to be achieved with an explicit loop. It is possible to implement repetition without a `GOTO` or any kind of looping statement, by using tail recursion.

A *tail recursion* is a recursive call to a function, in such a position within the function that it will always be the last thing executed before the function return. Frequently, this will be at the lexical end of that function, but it could also be nested within a control structure [Exhibit 12.3]. Tail recursion produces the same result as a simple infinite loop or simple backwards `GOTO`. Combined with a conditional, tail recursion can emulate a `WHILE` or `FOR` loop. The parallel between tail recursive functions and loops is so straightforward that either form can be mechanically translated to the other form, either by a programmer or by a compiler.

To use recursion to emulate a loop, one writes the body of the recursive function as an `IF` statement that tests the loop-termination condition. The answer returned by the function, if any, can be returned by the `THEN` clause, and the body of the loop is written as the scope of the `ELSE` clause. (A procedure that returns no value can put the loop body in the `THEN` clause and eliminate the `ELSE` clause.) The loop variable in the loop version is replaced by a parameter in the recursive version. On each recursive call, the parameter is increased or decreased by the value of the loop's increment expression.

The iterative and recursive mechanisms that implement repetition are quite different. In an iterative implementation, the contents of a single machine address, associated with the "loop variable" is changed during each repetition, and execution continues until the value of that variable reaches some goal. Exhibit 12.4 shows an iterative procedure to print the contents of part of an array. One storage object is allocated for the parameter `count`, and the value in that location is changed once for each item printed out.

Exhibit 12.3. Recursive implementation of repetition.

The following procedure prints the values of an integer array. We would call `print_out( LL, N );` to print out the first `N` items of array `LL`. The outer procedure is a shell that initializes a parameter for the inner, recursive procedure, `print_out_recursive`. The recursive procedure will be called `N` times and will create and initialize one copy of the parameter `count` each time. No copy of `count` is ever incremented. After the last item is printed, all `N` calls on `print_out_recursive` return, and all `N` copies of the parameters are deallocated.

```
Procedure print_out (list:  list_type; list_size:  integer);

Procedure print_out_recursive (count:integer);
Begin
    writeln( list [count] );
    if ( count < list_size ) then print_out_recursive (count+1);
End;

Begin
    print_out_recursive (1);
End;
```

In a tail-recursive implementation of repetition, each recursion causes allocation of additional storage on the stack for the current value(s) of the parameter(s) and for the function return address. New stack frames are piled on top of old ones. The recursive descent ends when the value of some parameter reaches a goal; thus this parameter must be different in each successive stack frame. When the value in the newest stack frame reaches the goal, repetition stops, and all the accumulated stack locations are freed, one frame at a time. Each parameter value is only used during the cycle that placed it on the stack. Values in locations below the top of the stack are never used again, and all are popped off together when the termination condition eventually happens.

A recursive version and an iterative version of a `print_out` procedure are given in Exhibits 12.3 and 12.4. Both routines are written in `Pascal`, and the implementations described are those produced by a typical `Pascal` compiler. These routines print out the elements of an array starting at the subscript `count`. The array, called `list`, has `list_size` elements. (To work correctly, `list_size` must be $>= 1$.) Let N be the number of items to be printed; $N =$ `list_size` $-$ `count` $+1$. The appropriate calls for the two inner procedures, to print out all list items, are identical:

```
print_out_recursive(1);
print_out_iterative(1);
```

The stack allocations for both ashown in Exhibit 12.5.

Repetition through recursion does use more storage space and more overhead time than repetition through iteration. Stack overflow could limit the number of repetitions. But coupled with some kind of conditional control structure and some form of sequential execution, recursion forms

Exhibit 12.4. Iterative implementation of repetition.

The following procedure uses iteration to print the values of an integer array. We would call `print_out( LL, N );` to print out the first `N` items of array `LL`. The outer procedure is a shell that initializes a parameter for the inner, iterative procedure, `print_out_iterative`. The iterative procedure will be called once. It creates one copy of the parameter `count`, which is incremented `N` times.

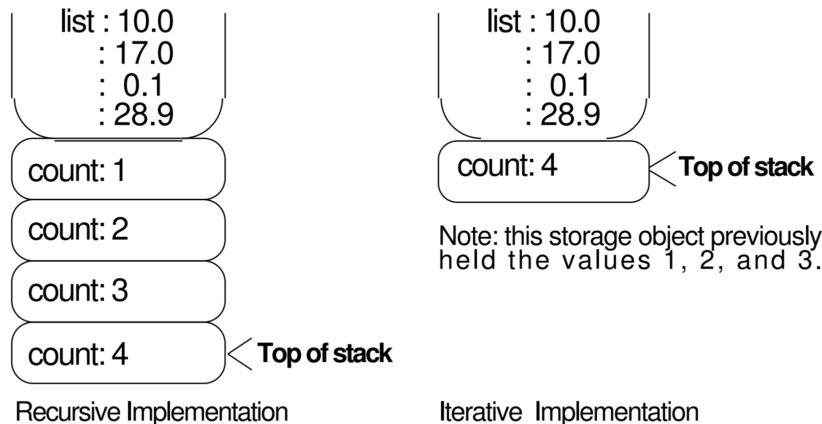
```
Procedure print_out (list:  list_type; list_size:  integer);

Procedure print_out_iterative (count:integer);
Begin
    While (count <= list_size) Do Begin
        writeln( list [count] );
        count := count + 1
    End
End;

Begin
    print_out_iterative (1);
End;
```

Exhibit 12.5. Stack diagrams for a tail recursion and a loop.

Assume that the global variable `list` is this array of four real numbers:

list →	10.0	17.0	0.1	28.9

The stacks for `print_out_recursive(1)` and `print_out_iterative(1)` are shown just as the last item on the list is being processed, but before the function return.



an adequate semantic basis for a programming language; explicit loops are not needed. Further, even when the programmer writes repetition using recursion, a translator can recognize tail recursion and generate iterative, not recursive, code. Translators for many languages descended from LISP do exactly that.

Whether or not to include iterative control structures in a language thus depends on some value judgements. The questions are:

- Is the iterative syntactic form better, more natural, or more appropriate than the recursive form for communicating some problems?
- Is iterative syntax better enough to warrant including nonnecessary statements in the language syntax?
- Will programmers be likely to write better code with the extra iterative forms?

If the answers are yes, iterative control structures should be included in a language. If not, they are excess baggage. The older LISP-like functional languages included explicit map functions which implement iteration over lists. Newer functional languages include extensive collections of list operators that do implied iteration.

12.2.3 Sequences

Traditional languages define the flow of control in terms of sequences of statements (control statements, procedure calls, and assignments). These languages also support nested expressions and function calls; some even include conditional expressions. The programmer often has a choice about whether to use heavily nested code, or to break that code into a sequence of statements. Writing a sequence of statements is often the simplest, clearest, and most direct way to express an idea. We have found by experience that pulling out the parts of a highly nested expression and writing them as a separate steps often clarifies the meaning of a program. It also permits the programmer to do a better job of attaching comments to obscure parts of the code.

Lambda calculus has shown us that statement sequences are really unnecessary in a theoretical sense. Everything can be done by nesting functions and function calls. However, lists of statements do provide the programmer with a second tool, in addition to function definitions, for breaking up code into comprehensible units. Modern functional languages meet the need for separating out parts of the code without supporting statement sequences at all. In this section, we examine how this is accomplished.

Nesting Replaces Assignment Sequences. A sequence of assignment statements *that does not assign two different values to the same variable* and, therefore, does not destroy information, can be emulated by a nest of functions with parameter bindings and local symbol definitions taking the place of the assignments. This is true in traditional languages as well as in functional languages.

Exhibit 12.6. A sequence of statements in Pascal.

```
c := a + b;
s := sin(c*c);
write (a, b, c, s);
```

Exhibit 12.6 shows a simple series of assignment and output statements, written in Pascal. In such a sequence, each line forms the context in which the next line is interpreted.

To form nested contexts in a functional language, one can use a nest of function calls and/or block declarations. Exhibit 12.7 shows how the Pascal assignment sequence might be rewritten in LISP using symbols defined in nested blocks. The first assignment in the Pascal sequence forms the outermost context block in LISP, creating the context for the rest of the nest. The final `write` in Pascal is innermost in the LISP version, so that its symbols are interpreted in the context of all the surrounding bindings.

Lazy Evaluation Replaces Sequences. In a Pascal program, the programmer must write statements physically in the same order that they will be evaluated. If statement B depends on the result of statement A, then A must come first. In contrast, when lazy evaluation is used, the sequence in which expressions are evaluated can be completely different from their order on the written page, because no expression is evaluated until its result is needed. When we look at this situation one way, we see that we have no direct control over the order of evaluation. When looked at from the other side, though, we see that we do not have to worry about that order!

Miranda provides the keyword `where` to permit the programmer to define and initialize local symbols. Local names and their defining expressions are written at the end of a Miranda definition, following `where`. These expressions are evaluated using lazy evaluation, an important difference from both Pascal and LISP. This frees the programmer from concern about the order in which the definitions are written within the `where` clause. Exhibit 12.8 defines a function called `Answers` that is analogous to the Pascal function in Exhibit 12.6. In `Answers`, the symbols `s` and `c` are both defined, but `s`, which depends on `c`, is defined first. This causes no problem. When `s` is used, its definition will be evaluated; in the middle of that process, where `s` refers to `c`, the expression for `c` will be evaluated. The order of the definitions on the page is immaterial.

Exhibit 12.7. A LISP function nest emulating a sequence.

```
(print (let ((c (+ a b)))
            (let ((s (sin (* c c))))
                 (list a b c s))))
```

Exhibit 12.8. Defining local symbols in Miranda.

This code is more or less like the Pascal and LISP examples in Exhibits 12.6 and 12.7. The function `Answers` returns a tuple of values, which will be printed if the function is called from the top level.

```
Answers a b = (a b c s)
              where
              s = sin(c * c)
              c = a+b
```

Note, also, that the explicit nesting present in LISP is gone. Miranda does not require each symbol to be defined before it is used. The set of `where` clauses, taken together, form a context for each other and for the function code. This is different from LISP semantics, which require that each clause be explicitly nested within the context formed by the definitions of all its symbols.

Guarded Expressions. In traditional languages, all statements must be written in one sequence or another. Frequently, though, the sequence of a short group of statements does not matter, so long as all are executed before the next part of the program. Some newer procedural languages, such as Concurrent Pascal[2] provide a *parallel execution* control structure, consisting of scope markers enclosing a series of statements. The semantics of parallel execution are that the statements in the scope may be executed in any sequence, or all at the same time, on different processors. No statement in such a scope computes a value that is referenced anywhere else in the group, and no two statements compute the same value.

This concept is carried over into the functional languages in the form of guarded expressions and `where` clauses. A *guarded expression* is a generalized conditional control structure in which the condition clauses are evaluated in parallel. It is very much similar to the generalized conditional in LISP. However, there is a critically important difference: the clauses of a LISP `cond` are evaluated sequentially, and the first true condition triggers evaluation of its corresponding clause. In contrast, a guarded expression is evaluated in *parallel* . This does not affect the outcome if only one guard is true. However, if two (or more) guards are true, either corresponding clause might be evaluated; a nondeterministic choice is made by the translator. Thus the value of a LISP conditional with more than one true clause is defined and deterministic because all clauses following the first true clause are ignored. In contrast, the value of a Miranda expression with two or more true guards is undefined. (Using such an expression is therefore risky.)

Output Sequences. The use of conditional expressions eliminates the need for conditional statements. Using recursion we can eliminate explicit loops. Using symbol definitions, nested contexts, and lazy evaluation we can eliminate sequences of assignments. Having done these things, the only remaining kinds of statements in traditional languages are those related to input and output.

[2] Brinch Hansen [1975].

Exhibit 12.9. A simple Miranda script.

Miranda	Lambda Calculus
`sq n = n * n`	`sq = λ n.  * n n`
`z = sq x / sq y`	`z = / (sq x) (sq y)`

The premise of lazy evaluation is that no expression is ever evaluated unless its result is needed. For most computations this works nicely, but it is not well suited to producing output. By definition, it is the side effect of the output process that is wanted, not the value returned. One device is to permit the programmer to specify that *strict evaluation* should be used. With strict evaluation all arguments are evaluated, even if they would be skipped using the rule for lazy evaluation. This permits the programmer to cause the evaluation of an expression whose result is not used by the rest of the program, and whose only effect is output.

Another way to produce an output sequence is to use the sequencing built into data objects in place of a sequence of statements. The compound data types, lists (sequences of values, all of the same type) and tuples (nonhomogeneous sequences of values), are both ordered sequences of values. Both can be constructed dynamically. Since structured objects are first-class objects in functional languages, any kind of output supported by the language (interactive output or file output) can also be used for lists and tuples.

12.3 Miranda: A Functional Language

Miranda is a simple and elegant functional language that illustrates both the principles and the attractiveness of the functional approach.[3] This section deals with the basic elements of Miranda; several advanced features are discussed in Chapter 18.

A program in Miranda is a collection of definitions, called a *script*, that define and name objects. Like lambda calculus, each definition has (on the left) the symbol being defined and (on the right) several clauses that comprise its definition. An object can be a simple data type, a list, a tuple, or a function. A very simple script which defines and calls the `square` function is shown in Exhibit 12.9.

A function call is written by writing the function name followed by one or more arguments, which can be simple items, lists, or tuples. No parentheses or delimiters of any kind are used to enclose the arguments. Note the similarity between the definition and calls on the function `sq`, shown on the left in Exhibit 12.9, and the corresponding definitions, expressed in lambda calculus, on the right. Note that Miranda uses infix notation for ordinary arithmetic operators and uses end-of-line rather than a character such as ";" to end the expression.

The primitive types in Miranda are characters (type `char`, written in single quotes), truth values (type `bool`, values `True` and `False`), and numbers, which include both integers and reals (type num).

[3] Turner [1986].

The basic data structures in Miranda are the list and the tuple. A list is an ordered sequence of homogeneously typed values and is written by enclosing the list values in square brackets. A string is a list of characters, and a literal string can be written as a list of characters, or as characters between double quote marks. A tuple is a nonhomogeneous sequence of values, and it is written by enclosing the values in parentheses.

12.3.1 Data Structures

Tuples

Miranda supports two compound data structures, tuples[4] and lists. A *tuple* is simply a list of objects whose types do not have to be alike. It could be implemented by a record data type or by a list of pointers to the members of the tuple. In Miranda we denote a literal tuple by enclosing a series of values in parentheses. The argument list of a function is also a tuple, although it is written with no punctuation at all.

Tuples are used in Miranda to define enumerated data types and to form complex data structures, just as records are used in traditional languages. The methods for defining and accessing tuple types involve a powerful pattern-matching mechanism. They are dealt with in Chapter 18.

Lists

Miranda lists can be denoted in several ways. The simplest list is a sequence of data items enclosed in square brackets [Exhibit 12.10]. However, this notation is cumbersome for long lists and impossible for infinite lists, so Miranda also provides shorthand ways to denote lists. We can denote a list of consecutive values by giving the first and last value. Thus [1..5] means the same thing as [1,2,3,4,5]. This notation can be extended to denote any arithmetic progression of values. The first two values are written, followed by ".." and the last value. This is enough information to define the list; the difference between the first value and second value is taken as the step size for the progression. An *infinite list*, such as "all the numbers greater than 10", is denoted by giving just the first value or the first two values. A list may also be denoted by a powerful kind of expression known as a list comprehension, which is discussed below.

12.3.2 Operations and Expressions

The usual arithmetic operators (`+, -, *. /, div, mod`) are defined with the usual precedence and are written in infix notation. Miranda has some basic operators for building and decomposing lists, as well as several powerful operations that work on list arguments. Exhibit 12.11 shows the use of the five basic list operators, which are:

[4]"Tuple" rhymes with "scruple", and it comes from the mathematical term "n-tuple", which comes from the family of words "quadruple", "quintuple", etc.

Exhibit 12.10. Denoting lists in Miranda.

```
[]                              The null list.
[1..n]                          The integers from one to n.
odd_numbers = [1,3..100]        Odd numbers from one to one hundred.
evens = [ 10,12 ..  100]        The even numbers between 10 and 100.
eleven_up = [11 ..  ]           All numbers greater than 10.
evens_up = [12,14 ..]           All even numbers greater than 10.
week_days = ["Mon","Tue","Wed","Thur","Fri"]
                                A list of 5 strings is bound to week_days.
```

L1 ++ L2	Concatenate list L2 onto the end of L1.
item : List	Add the item to the head of the List.
List ! n	Select the nth item from the List.
L1 -- L2	Remove values in L2 from list of values in L1.
# List	Return the number of items on the List.

12.3.3 Function Definitions

A function definition is called an "equation" because it does look like a mathematical equation. The simplest functions can be written on one line, like the function `sq` in Exhibit 12.9. The function name is written, followed by a dummy parameter list, followed by "=" and an expression which is written in terms of the dummy parameters. If the definition requires more than one line of code, succeeding lines must be indented.

In Miranda, guarded expressions are used for conditional control. A guarded expression consists of a series of lines, each with an expression on the left, followed by a comma and a Boolean guard on the right. The interpretation is this: The guard expressions are evaluated in parallel. If the result of any guard is true, the corresponding expression is evaluated and returned. If more than

Exhibit 12.11. Using list operations in Miranda.

Expression	Result
`week_days = ["Mon","Tue","Wed","Thur","Fri"]`	
`# week_days`	5
`days = week_days ++ ["Sat","Sun"]`	Make a list of 7 days.
`0:[1,2,3]`	`[0,1,2,3]`
`week_days !  2`	`"Wed"` (Zero-based subscript.)
`week_day -- "Fri"`	`["Mon","Tue","Wed","Thur"]`

Exhibit 12.12. A guarded expression.

```
gcd a b = gcd (a-b) b,    a>b
        = gcd a (b-a),    a<b
        = a,              a=b
```

one guard is true, any one of the corresponding values might be returned. An "otherwise" clause is permitted at the end.

A recursive definition of the greatest common divisor algorithm, implemented using a guarded expression, is shown in Exhibit 12.12. This shows the proper use of guards—exactly one guard is true, and the set of guards explicitly cover all possibilities.

In a functional language, binding must take the place of assignment. Of course, a function call offers an opportunity for evaluating expressions and binding the results to parameter names. However, a nest of many function calls can be completely unreadable and difficult for the programmer to construct. Miranda provides parameter binding plus a second way to bind a value to a name: the `where` clause. Within a function definition, "`where`" is used to define a local name for the value of an expression. The Miranda programmer uses `where` in the same way that a Pascal programmer uses assignment, to give a name to a subexpression that will be used more than once. `Where` clauses may be nested to arbitrary depth, producing block structured name binding. Indentation of inner blocks is compulsory, because layout information is used by the parser.

Compare the Miranda code in Exhibit 12.13 to the more-or-less equivalent Pascal code in Exhibit 12.14. Both functions compute the two roots of a quadratic equation, but the Pascal code is twice as long and far more complex. This is due to several factors:

- Guarded expressions are syntactically briefer than nested `if...then...else` statements.
- Miranda frees the programmer from concern about evaluation order. The local symbols are

Exhibit 12.13. `Where` **takes the place of assignment.**

```
quadroot a b c = error "complex roots",       delta < 0
               = [term1],                     delta = 0
               = [term1+term2, term1-term2],  delta > 0
                 where
                 delta = b*b - 4*a*c
                 radix = sqrt delta
                 term1 = -b/(2*a)
                 term2 = radix/(2*a)
```

Exhibit 12.14. Assignment used to name subexpressions in Pascal.

```
type root_list= ^root_type;
type root_type = record  root:  real;  next:  root_list  end;

function quadroot(a, b, c:  real):  root_list;
var delta, radix, term2:  real;
    roots, temp:  root_list;
begin
    roots := NIL;
    delta := b*b - 4*a*c;
    if delta < 0
    then writeln('Error:  complex roots')
    else begin
        new(roots);
        roots^.root:= -b/(2*a);
        if delta = 0
        then roots^.next := NIL
        else begin
            radix := sqrt(delta);
            term2 := radix/(2*a);
            new (temp);
            roots^.next:= temp;
            temp^.root := roots^.root - term2;
            roots^.root := roots^.root + term2;
            temp^.next := NIL
        end
    end;
    quadroot := roots
end;
```

Exhibit 12.15. The reduce function in Miranda.

The reduce function may be defined in Miranda to take three parameters: a function, a list, and a number. The third argument is the identity value for the given function; its value is used to initialize the summing process and is returned as the value of `reduce` when the list argument is a null list.

```
reduce f [] n = n
reduce f (a:x) n = f a (reduce x n)
```

The type of `reduce`, as discussed in Chapter 9, Section 9.3, is:

```
reduce ::  (num → num → num )
          → [num]
          → num
          → num
```

defined in a group at the bottom. They will be evaluated if needed and in whatever order they might be needed. The order in which the `where` clauses are written is immaterial.

- The Pascal code uses nested conditionals to implement lazy evaluation of the local variables. In Miranda, lazy evaluation just happens—the programmer does not have to force it by carefully ordering and nesting the code. If a subexpression is not needed, its `where` clause is not evaluated.

- The goal of implementing denotational philosophy forced Miranda's designers to provide an assignment-free method for returning a list. The result is that Miranda lets the programmer denote variable-length lists with ease; the programmer simply writes a list of expressions within brackets. In Pascal, however, the list must be constructed and initialized piece by painful piece.

Higher-order Functions

Like all functional languages, Miranda supports higher-order functions. Thus Miranda programs often use partial parameterization through closures and functional parameters. A Miranda version of the "reduce" function is given in Exhibit 12.15; compare its simplicity to the Pascal version in Exhibit 9.29.

12.3.4 List Comprehensions

Loops that use explicit loop variables are inherently nonfunctional; both `FOR` and `WHILE` loops depend on statement sequences and assignment. In contrast, a *comprehension* is a pure functional construct which provides a way to specify iteration without using assignment or sequences

Exhibit 12.16. The syntax for a list comprehension.

ZF expr ::=	'[' ⟨body⟩ '|' ⟨qual-list⟩ ']'
qual-list ::=	generator { generator | filter }
generator ::=	⟨variable⟩ ← ⟨list-expression⟩
filter ::=	⟨Boolean expression⟩
body ::=	⟨list expression⟩

of statements.

A *list comprehension* is an expression that denotes a finite or infinite list with an arbitrarily complex structure.[5] A comprehension has two parts: on the left is an expression. and on the right are a series of clauses called a *qualifier list*, consisting of generators and filters separated by semicolons. The qualifier list specifies an ordered series of values to bind to the free variable(s) in the expression. The syntax for list comprehensions is shown in Exhibit 12.16 and an example is shown in Exhibit 12.17. The clauses that form the qualifier list consist of one or more list-expressions called "generators" followed by zero or more Boolean expressions called "filters".[6]

In many ways a comprehension is like a `FOR` loop control structure. The expression is like the body of the loop, its free variable(s) are like the loop control variable(s), and the generators and filters specify a series of values for the free variable, much as a loop control element specifies a series of values for the loop variable. A comprehension differs from a loop because a comprehension returns a list as its result, whereas a loop does not return a value. A comprehension with one generator is like a single loop; with two generators (for two free variables) a comprehension is like a pair of nested loops.

For simplicity, let us first examine simple comprehensions with just one generator. A generator is used to produce and name an ordered set of values. It specifies a symbol and a list expression. The list expression generates a series of values which are bound, in sequence, to the symbol. Then the symbol with its value is passed through all of the filters; if any filter evaluates to FALSE, the value is discarded and the next value is generated. If the value passes all the tests, it is substituted for its variable in the expression on the left side of the comprehension and used to compute the

[5]This terminology is from Haskell; the term used in Miranda is "Zermelo Frankel expression", or ZF expression. This form was named after two early logicians who developed set theory.

[6]The use of generators and filters goes back to an early low-level language named IPL/5, Newell [1961]. Notation similar to that shown here was used by Schwarz in his set-processing language, SetL.

Exhibit 12.17. A list comprehension.

Expression: `[ n+2 | n ← [1..10] ]`

Read this as: the list of all values for `n+2` such that `n` takes on the list of values 1 through 10.

Exhibit 12.18. Some list comprehensions.

	Expression	Result
1	`[ n*2 | n ← [2 .. 10] ]`	`[4, 8, 12, 16, 20]`
2	`[ n*n | n ← [1, 2 .. ]]`	All perfect squares; 1, 4, 9, etc.).
3	`[ x+y | x ← [1..3]; y ← [3, 7] ]`	[4, 5, 6, 8, 9, 10]
4	`[ x*y | x ← [1..3]; y ← [1..3]; x>=y ]`	[1, 2, 4, 3, 6, 9]

next value in the output list.

Let us look at a series of list comprehensions to see how generators and filters work. Line 1 in Exhibit 12.18 is a very simple example. A single generator is used to produce a list of even numbers from 2 to 10. This list is "fed into" the expression on the left, producing a list of multiples of 4 from 4 to 20. In line 2, the result is an infinite list of square numbers.

The combination of generators and filters can specify a list whose structure is considerably more complex than the control element of a `FOR` loop, because a generator can start with any existing list, and any number of filters can be applied to the elements on that list. To use a comprehension, the programmer specifies a list of values that contain all the ones of interest and perhaps some other, unwanted values. Then filters are appended to specify subsets of values to remove from the result. Several filters may be needed if the values to be excluded form a complex pattern.

A comprehension with two free variables needs two generators. In this case, the generators are executed in a nested fashion; the second generator runs through all of its values for each value generated by the first. The number of value-pairs generated is the product of the lengths of the two generator lists. In line 3, two lists are generated and the final result is the pairwise sum of those lists. Even though two generators are nested here, the final result is a linear list, not a multidimensional structure. A filter may test just one member of the value-pair, or it may combine them in a test [Exhibit 12.18, line 4]. Thus the length of the output from a comprehension can be anything from zero up to the product of the lengths of all the generated lists.

When list comprehensions are used in a program, the result is often a concise, clear statement of the algorithm. For example, consider the quicksort program in Exhibit 12.19. The function `sort` takes a list as its argument and returns a sorted list. Line 1 defines the action of `sort`

Exhibit 12.19. Quicksort in Miranda.

```
1.  sort [] = []
2.  sort (pivot:rest) = sort [ y | y ← rest; b ≤ pivot ]
3.                      ++ [pivot] ++
4.                      sort [ z | z ← rest; b > pivot ]
```

on a null-list argument—it simply returns the null list. This is the base case for the recursive definition of `sort`. Lines 2 through 4 define `sort` on all other lists. The fact that the argument must be a list is denoted by the parentheses between the function name and the "`=`" sign which terminates the header. (Remember, parentheses are not used to enclose parameter lists.) Within these parentheses, the ":" notation is used to give local names to the head and tail of the argument. Thus the name "`pivot`" will be bound to the first element, or head, of the list, and "`rest`" will be bound to the rest (or tail) of the list.

The `sort` function works by separating its argument (a list) into three sections: the first element, named `pivot`, one new list containing all elements from the rest of the list that are less than or equal to the `pivot` (line 2), and a second new list containing all elements that are greater than the `pivot` (line 4). The result returned by `sort` is these three parts, concatenated (lines 2 through 4) in the correct order, after the sublists have been recursively sorted.

The program differs from a quicksort program written in a procedural language because it does *not* define the procedure for finding "all the elements of the list that are less than the first element, pivot". Instead, a list comprehension is used to *denote* those elements and supply them as an argument to a recursive call on the sort function. Implementation of the actual process of finding those elements is left to the system.

This definition of quicksort is certainly brief and is relatively easy to understand. Further, a proof of its correctness is straightforward because the program is simply a statement of what it means to be "sorted". However, note the inefficiency involved in processing each element of the input list twice, once to find small elements, and again to find large ones!

12.3.5 Infinite Lists

Some of the lists denoted in Exhibits 12.10 and 12.18 are infinite lists; that is, they go on forever. In this section we examine how an infinite list can be represented in a real computer, and why we might want to use one.

Think of an infinite list as having two parts, a finite head, consisting of values that have already been computed, and an infinite tail, consisting of code for generating the rest of the list. We will be lazy about constructing this list—new values will only be computed when they are needed. The program may access this list like any other, for example, by using subscripts. When the program needs a value at the head of the list, the desired value is found and returned.

The infinite tail of the list will not be represented as data (obviously). Instead, it will be a functional object. When the program needs a value on the tail of the list, the function on the tail is used to extend the head of the list as follows. When it is called, it computes the next list item and constructs a new functional object for computing the rest of the list. The value and the function become the new tail of the list, replacing the old functional object. (The list has now grown by one data item.) This process is repeated until the required list item is generated.

These functional objects can be implemented by closures, consisting of a context that remembers the current position in the list, and a function that, given the current position, can calculate the next list item. The new functional object that is created is just a new closure, whose context is the

Exhibit 12.20. A memoized function.

```
pow10 0       = 1
pow10 (n+1)   = 10 * powlist !  n

powlist = [ pow10(x)  |  x <- [0..]  ]
```

new position on the list.

Memo Functions.

A generally useful device that can be used in any language is called *memoizing*. We do it to increase efficiency when a function is called repeatedly on the same inputs. For example, consider the process of ASCII-to-floating-point number conversion, which must be done each time a floating-point number is read from the input stream. The last step in the conversion is to divide the intermediate result by a power of 10 that corresponds to the number of decimal places in the input. We can write a function `pow10(d)` that returns 10 to the power `d`. If many numbers are to be read, though, we would expect to need the same result many times, and computing powers is a slow job. The obvious answer is to make a table of powers of 10 ahead of time, then just use a subscript to extract the right power. We can conceive of this in a lazy way. How many powers of 10 do we need? That isn't really known ahead of time. Should we just calculate a bunch and hope that our table is long enough? What happens if it isn't? Infinite lists provide a straightforward solution to this problem. We can use an infinite list to memoize our `Pow10` function. When a particular power of 10 is needed for the first time, the infinite list will be extended that far (and no farther). If a power is called for that is already on the list, it will be returned directly. Exhibit 12.20 shows a version of this function memoized by using an infinite list.

Input and Output

Input and output are obvious necessities for any programming language, including a functional language. However, both are inherently time-dependent. The usual concept of input, that a storage object takes on a series of values brought in by executing READ requests, is not consistent with the functional approach. However, infinite lists provide a functionally acceptable way to model I/O.[7] The input stream is modeled as an infinite list of values. Each time you poke the input stream, a new value is appended to the list of inputs and can then be accessed by a program.

We can also model the output stream as an infinite list. It is important, though, that real output starts to happen before the entire list is evaluated. This can be guaranteed, even in a lazy system, by having the output function return a "success" or "fail" response and then testing that

[7]Continuations can also be used to implement I/O.

response. The act of testing forces the output function to be evaluated, which has the side effect of making the output happen.

Exercises

1. Explain what it means to denote a function as opposed to writing code for a function.
2. Define "mutability" and explain why it is undesirable.
3. Explain the sense in which a Pascal or C programmer must "over-sequence" code.
4. How does lazy evaluation eliminate much of the concern about sequencing?
5. How does recursion differ from iteration?
6. Name three mechanisms or data structures that must be part of the semantic basis of a language that supports recursion.
7. FORTRAN supports local variables and parameters in functions. What semantic mechanism is missing that makes FORTRAN unable to support recursion?
8. What is tail recursion?
9. How can recursion be used to eliminate the need for the `WHILE` construct?
10. What is the minimal set of necessary control structures? Name a language that contains these alone.
11. What is the difference between lazy and strict evaluation?
12. What is a guarded expression? How can it be misused?
13. What is the difference between a tuple and a list in Miranda?
14. What is an infinite list? How can it be used to model I/O?
15. What is a list comprehension? Qualifier? Filter?
16. How does a list comprehension differ from a traditional loop?
17. What is memoizing? Why is it used?
18. Write a Miranda script for Euclid's greatest common divisor algorithm. Compare your code to the `gcd` programs in Prolog [Exhibit 13.13] and C [Exhibit 13.14].