**UNIT-V**

**Functional programming languages** **7**

So far in this book we have been concerned primarily with languages which may be described as statement-oriented or imperative. These languages are affected strongly by the architecture of conventional computers. Functional programming languages take as their basis not the underlying computing engine, but rather the theory of mathematical functions. Rather than efficient execution, these languages are motivated by the questions: what is the proper unit of program decomposition and how can a language best support program composition from independent components.

We have seen that procedural languages use procedures as the unit of pro-gram decomposition. Procedures generally use side effects on global data structures to communicate with other procedures. Abstract data types attempt to modularize a program by packaging data structures and operations together in order to limit the scope of side-effects. Functional programs reduce the impact of side-effects further, or even eliminate them entirely, by relying on mathematical functions, which operate on *values* and produce *values* and have no side-effects.

We start in the next section by describing the main elements of imperative programming. These elements help illustrate the main differences with func-tional programming. To contrast these differences further, we will then com-pare mathematical functions with programming langauge functions. In Section 5.3.2 we present Lambda calculus as a model for function definition,

evaluation and composition. We then look at ML and LISP as examples of functional programming languages. Early functional languages, starting from LISP, were dynamically typed and scoped. Scheme is a dialect of |LISP that introduces static scoping into the language. Later functional languages, such as ML, not only include static scoping but also static typing. Many functional languages, include both Scheme and ML, have also added a module construct to address programming in the large.

**5.1 Characteristics of imperative languages**

Imperative languages are characterized by three concepts: variables, assign-ment, and sequencing. The state of an imperative program is maintained in program variables. These variables are associated with memory locations and hold values and have addresses. We may access the value of a variable either through its name (directly) or through its address (indirectly). The value of a variable is modified using an assignment statement. The assignment state-ment introduces an order-dependency into the program: the value of a vari-able is different before and after an assignment statement. Therefore, the meaning (effect) of a program depends on the order in which the statements are written and executed. While this is natural if we think of a program being executed by a computer with a program counter, it is quite unnatural if we think of mathematical functions. In mathematics, variables are bound to val-ues and once bound, they do not change value. Therefore, the value of a func-tion does not depend on the order of execution. Indeed, a mathematical function defines a mapping from a value domain to a value range. It can be viewed as a set of ordered pairs which relate each element in the domain uniquely with a corresponding element in the range. Imperative programming language functions, on the other hand, are described as algorithms which specify how to compute the range value from a domain value with a pre-scribed series of steps.

One final characteristic of imperative languages is that repetition— loops— are used extensively to compute desired values. Loops are used to scan through a sequence of memory locations such as arrays, or to accumulate a value in a given variable. In contrast, in mathematical functions, values are computed using function application. Recursion is used in place of iteration. Function composition is used to build more powerful functions.

Because of their characteristics, imperative languages have been given labels such as state-based and assignment-oriented. In contrast, functional languages have been called value-based and applicative.

**5.2 Mathematical and programming functions**

A function is a rule for mapping (or associating) members of one set (the do-main set) to those of another (the range set). For example, the function “ square” might map elements of the set of integer numbers to the set of inte-ger numbers. A function definition specifies the domain, the range, and the mapping rule for the function. For example, the function definition

square(x) ≡ x\*x, x is an integer number

defines the function named “ square” as the mapping from integer numbers to integer numbers. We use the symbol “ ≡” for “ is equivalent to.” In this defini-tion, *x* is a *parameter*. It stands for *any* member of the domain set.

Once a function has been defined, it can be *applied* to a particular element of the domain set: the application yields (or *results* in, or *returns*) the associated element in the range set. At application time, a particular element of the domain set is specified. This element, called the *argument*, replaces the parameter in the definition. The replacement is purely textual. If the defini-tion contains any applications, they are applied in the same way until we are left with an expression that can be evaluated to yield the result of the original application. The application

square (2)

results in the value 4 according to the definition of the function square.

The parameter *x* is a mathematical variable, which is not the same as a pro-gramming variable. In the function definition, x stands for any member of the domain set. In the application, it is given a specific value— *one* value. Its value never changes thereafter. This is in contrast to a programming variable which takes on different values during the course of program execution.

New functions may be created by combining other functions. The most com-mon form of combining functions in mathematics is function composition. If a function F is defined as the composition of two functions G and H, written as

F ≡ G o H,

applying F is defined to be equivalent to applying H and then applying G to the result.

In conventional programming languages, a function is defined proceduraly: the rule for mapping a value of the domain set to the range set is stated in terms of a number of steps that need to be “ executed” in certain order speci-fied by the control structure. Mathematical functions, on the other hand, are defined applicatively— the mapping rule is defined in terms of combinations or applications of other functions.

Many mathematical functions are defined recursively, that is, the definition of the function contains an application of the function itself. For example, the standard mathematical definition of factorial is:

n! ≡ **if** n = 0 **then** 1 **else** n \* (n - 1)!

As another example, we may formulate a (recursive) function to determine if a number is a prime:

prime (n) ≡ **if** n = 2 **then** true **else** p (n, n **div** 2)

where function p is defined as:

p (n, i) ≡ **if** (n **mod** i) = 0 **then** false

**else if** i = 2 **then** true

**else** p (n, i - 1)

Notice how the recursive call to p(n, i-1) takes the place of the next iteration of a loop in an imperative program. Recursion is a powerful problem-solving technique. It is a used heavily when programming with functions.

**5.3 Principles of functional programming**

A functional programming language has three primary components:

1. A set of data objects. Traditionally, functional programming languages have provided a single high level data structuring mechanisms such as a list or an array.
2. A set of built-in functions. Typically, there are a number of functions for manipulating the basic data objects. For example, LISP and ML provide a number of functions for building and accessing lists.
3. A set of functional forms (also called high-order functions) for building new functions. A common example is function composition. Another common example is function reduction. *Reduce* applies a binary function across successive elements of a sequence. For example, reducing + over an array yields the sum of the elements of the array and reducing

\*over the elements of an array yields the product of the elements of the array. In APL, 

is the reduction functional form (called operator in APL) and it takes one operation as argument. The plus reduction can be accomplished by /+ and the multiplication reduction by /\*. The use of functional forms is what distinguishes a functional program. Functional forms support the combination of functions without the use of control structures such asses iteration conditional statements.

The execution of functional programs is based on two fundamental mecha-nisms: binding and application. *Binding* is used to associate values with names. Both data and functions may be used as values. Function application is used to compute new values.

In this section we will first review these basic elements of functional pro-grams using the syntax of ML. We will then introduce Lambda calculus, a simple calculus that can be used to model the behavior of functions by defin-ing the semantics of binding and application precisely.

**5.3.1 Values, bindings, and functions**

As we said, functional programs deal primarily with values, rather than vari-ables. Indeed, variables denote values. For example 3, and “a ” are two con-stant values. A and B are two variables that may be *bound* to some values. In ML we may bind values to variables using the binding operator =. For exam-ple

val A = 3;

val B = "a";

The ML system maintains an environment that contains all the bindings that the program creates. A new binding for a variable may hide a previous bind-ing but does not replace it. Function calls also create new bindings because the value of the actual parameter is bound to the name of the formal parame-ter.

Values need not be just simple data values as in traditional languages. We may also define values that are functions and bind such values to names:

val sq = fn(x:int) => x\*x;

sq 3;

will first bind the variable sq to a function value and then apply it to 3 and print 9. We may define functions also in the more traditional way:

fun square (n:int) = n \* n;

We may also keep a function anonymous and just apply it without binding it to a name:

(fn(x:int) = x\*x) 2;

We may of course use functions in expressions:

2 \* sq (A);

will print the value of the expression 2A2.

The role of iteration in imperative languages is played by recursion in func-

int fact(int n)

{ int i=1;

assert (n>0);

{for (int j=n; j>1; ++j)

i= i\*n;

return i;

fun fact(n) =

if n = 0 then 1

else n\*fact(n-1);

}

**FIGURE 84.**Definition of factorial in C++ and ML

The role of iteration in imperative languages is played by recursion in functional languages. For example, Figure 84 shows the function factorial written in C++ using iteration and ML using recursion.

We saw in Chapter 4 that functions in ML may also be written using patterns and case analysis. The factorial program in the figure may be written as com-posed of two cases, when the argument is 0 and when it is not:

fun fact(n) =

fact(0) = 1

| n\*fact(n-1);

In addition to function definition, functional languages provide functional forms to build new functions from existing functions. We have already men-tioned mathematical function composition operator o as such a higher order function. It allows us to compose two functions F and G and produce a new function FoG. Functional programming languages provide both built-in higher order functions and allow the programmer to define new ones. Most lan-guages provide function composition and reduction as built-in functional forms.

**5.3.2 Lambda calculus: a model of computation by functions**

In the previous section, we have seen the essential elements of programming with functions: binding, function definition, and function application. As opposed to an imperative language in which the semantics of a program may be understood by following the sequence of steps specified by the program,the semantics of a functional program may be understood in terms of the computation implied by function applications. Lambda calculus is a surpris-ingly simple calculus that models the computational aspects of functions. Studying lambda calculus helps us understand the elements of functional pro-gramming and the underlying semantics of functional programming lan-guages independently of the syntactic details of a particular programming language.

Lambda *expressions* represent values in the lambda calculus. There are only three kinds of expressions:

1. An expression may be a single identifier such as x.
2. An expression may be a function definition. Such and expression has the form λ x.e which stands for the expression e with x designated as a *bound* variable. The expression e represents the body of the function and x the paramter. The expression e may contain any of the three forms of lambda expressions. Our familiar square function may be written as λ x.x\*x.
3. An expression may be a function application. A function application has the form e1 e2 which stands for expression e1 applied to expression e2. For example, our square function may be applied to the value 2 in this way: ( λ x.x\*x) 2. Informally, the result of an application can be derived by replacing the parameter and evaluating the resulting expression.

( λ x.x\*x) 2=

2\*2 =4

In a function definition, the parameters following the " λ " and before the "." are called *bound* variables. When the lambda expression is applied, the occur-rences of these variables in the expression following the “ .” a re replaced by the arguments. Variables in the definition that are not bound are called *free* variables. Bound variables are like local variables, and free variables are like nonlocal variables that will be bound at an outer level.

Lambda calculus capture the behavior of functions with a set of rules for rewriting lambda expressions. The rewriting of an expression models a step in the computation of a function. To apply a function to an argument, we rewrite the function definition, replacing occurrences of the bound variable by the argument to which the function is being applied.

Thus, to define the semantics of function application, we first define the con-cept of *substitution*. Substitution is used to replace all occurrences of an iden-tifier with an expression. This is useful to bind parameters to arguments and to avoid name conflicts that arise if the same name appears in both the expression being applied and the argument expression to which it is being applied. We will use the notation [e/x]y to stand for “ substitute e for x in y.” We will refer to variables as xi. Two variables xi and xj are the same if i=j. They are not the same if i=/=j. We can define substitution precisely with the following three rules, based on the form of the expression y:

1. If the expression is a single variable: [e/xi]xj= e, if i = j
	* xj, if i=/= j
2. If the expression is a function application, we first do the substitution both in the function definition and in the argument, and then we apply the resulting function to the resulting argument expression:

[e1/x](e2 e3)= ([e1/x]e2)([e1/x]e3)

In doing the substitutions before the function application, we have to be careful not to create any bindings that did not exist before or invalidate any previous bindings. This means that we may not rename a variable and make it bound if it were free before or make it free if it were bound before. The next rule takes care of these situations.

1. If the expression is a function definition, we must do the substitution carefully: [e1/xi]( λ xj.e2)= λ xj.e2, if i=j
	* + λ xj.[e1/xi]e2, if i=/=j and xj is not free in e1 (otherwise, it would become

newly bound)

* + - λ xk.[e1/xi]([xk/xj]e2), otherwise, where k=/=i, k=/=j, and xk is not free in either e1 or e2

The last rule serves to rename all occurrences of a variable by another name to avoid name clashes.

Using the substitution rules above, we can define the semantics of functional computations in terms of rewrite rules. That is, we define the result of a func-tion application in terms of rewriting the definition of the function, replacing the bound variables of the function with corresponding arguments. The fol-lowing three rewrite rules define the concept of function evaluation:

1. Renaming: λ xi.e <=> λ xj.[xj/xi]e, where xj is not free in e. The renaming rules says that we can replace all occurrences of a bound variable with another name without affecting the meaning of the expression. In other words, a function is abstracted over the bound variables.
2. Application: ( λ x.e1)e2 <=> [e2/x]e1. This rule says function application means replacing the bound variable with the argument of the application.

3. λ x.(e x) <=> e, if x is not free in e.

The last rule says that free variables are the only way for an environment to change the effect of a function. That is, a function is a self-contained entity with the parameters being its only interface.

These rules may be used in the forward direction to “ reduce” a lambda expression. In fact, any lambda expression may be reduced using these three rules until no further reduction is possible. An expression that may no longer be reduced is said to be in *normal form*.

For example, the following shows the application of the three rules to reach a normal form for the original expression.

( λ x.( λ y.x+y) z) ( λ y.y\*y) =

( λ x.x+z)) ( λ y.y\*y) =

( λ y.y\*y)+z

The simple semantics that we have described here capture the semantics of binding, function definition and function application, which are the primitive elements of functional programming languages. The clear semantics of func-tional languages is due to the fact that the semantics of function definition and application can be defined with these three simple rules.

One of the interesting aspects of lambda calculus is that we can define the semantics of functions using only one-argument functions. To deal with func-tions of more than one argument, a list of arguments is passed to the function f which applies to the first argument and produces as result a function that is then applied to the second argument, and so on. This technique is called *cur-rying* and a function that works this way is called a *curried function*.

For example, consider a function to sum its two arguments. We could write it as λ x,y.x+y. This is a function that requires two arguments. But we could also write it as λ x. λ y.x+y. This new function is written as the composition of two functions, each requiring one parameter. Let us apply it to arguments 2 3:

( λ x. λ y.x+y) 2 3 =

(( λ x. λ y.x+y) 2) 3 =

( λ y.2+y) 3 =

2+3 =

5



f x y z.

This is a common technique in functional programming to deal with a vari-able number of arguments. Each argument is handled in sequence through one function application. Each function application replaces one of the bound variables, resulting in a “ partially evaluated” function that may be applied again to the next argument. Symbolically, (f x y z) is considered to be (((f(x)) y) z). Indeed, in ML, the function application f(x,y,z) may also be written in the curried form

**5.4 Representative functional languages**

In this section, we examine pure LISP, APL, and ML. LISP was the first functional programming language. The LISP family of languages is large and popular. LISP is a highly dynamic language, adopting dynamic scoping and dynamic typing, and promoting the use of dynamic data structures. Indeed garbage collection was invented to deal with LISP’s heavy demands on dynamic memory allocation. One of the most popular descendants of LISP is Scheme, which adopts static scope rules.

APL in an expression-oriented language. Because of the value-orientation of expressions, it has many functional features. As opposed to LISP’s lists, the APL data structuring mechanism is the multidimensional array.

ML is one of the recent members of the family of functional programming languages that attempt to introduce a strong type system into functional pro-gramming. We will examine ML in more detail because of its interesting type structure. In the next section, we look at C++ to see how the facilities of a conventional programming language may be used to implement functional programming techniques.

Most functional programming languages are *interactive*: they are supported by an interactive programming system. The system supports the immediate execution of user commands. This is in line with the value-orientation of these languages. That is, the user types in a command and the system immedi-ately responds with the resulting value of the command.

**5.4.1 ML**

ML starts with a functional programming foundation but adds a number of features found to be useful in the more conventional languages. In particular, it adopts polymorphism to support the writing of generic components; it adopts strong typing to promote more reliable programs; it uses type infer-ence to free the programmer from having to make type declarations; it adds a module facility to support programming in the large. The most notable contri-bution of ML has been in the area of type systems. The combination of poly-morphism and strong typing is achieved by a “ type inference” mechanism used by the ML interpreter to infer the static type of each value from its con-text.

*5.4.1.1 Bindings, values, and types*

We have seen that establishing a binding between a name and a value is an essential concept in functional programming. We have seen examples of how ML establishes bindings in Section 5.3.1. Every value in ML has an associ-ated type. For example, the value 3 has type int and the value fn(x:int) =>x\*x has type int->int which is the signature of the functional value being defined.

We may also establish new scoping levels and establish local bindings within these scoping levels. These bindings are established using let expressions:

let x = 5

in 2\*x\*x;

evaluates to 50. The name x is bound only in the expression in the let expres-sion. There is another similar construct for defining bindings local to a series of other declarations:

local

x = 5

in

val sq = x\*x

val cube = x\*x\*x

end;

Such constructs may be nested, allowing nested scoping levels. ML is stati-cally scoped. Therefore, each occurrence of a name may be bound statically to its declaration.

*5.4.1.2 Functions in ML*

In ML, we can define a function without giving it a name just as a lambda expression. For example, as we have seen:

fn(x, y):int => x\*y

is a value that is a function that multiplies its two arguments. It is the same as the lambda expression λ x,y.x\*y. We may pass this value to another function as argument, or assign it to a name:

val intmultiply = fn(x, y):int => x\*y;

The type of this function is fn:int\*int->int.

We have seen that functions are often defined by considering the cases of the input arguments. For example, we can find the length of a list by considering the case when the list is empty and when it is not:

fun length(nil) = 0

| length([\_::x]) = 1+length(x);

The two cases are separated by a vertical bar. In the second case, the under-score indicates that we do not care about the value of the head of the list. The only important thing is that there exists a head, whose value we will discard.

We may also use functions as values of arguments. For example, we may define a higher-order function compose for function composition:

fun compose (f, g)(x) = f(g(x));

The type of compose is (’a->’b \* ’c->’a)->(’c->’a). ML provides some built-in functional forms as well. The classic one is map which takes two arguments, a function and a list. It applies the function to each element of the list and forms the results of the applications into a list. For example, the result of:

val x= map (length,[[], [1,2,3],[3]);

is [0,3,1].

For example, the reduce function takes a function F of two arguments and a nonempty list [a1,a2,...,an] as arguments and produces as result the value F(a1, F(...F(an-1,F(an))). The basis case F(x) applied to a singleton list is defined to be the singleton element. So, the result of

val x = reduce(+, [1,2,3,4]);

is 10. Some systems provide the function reduce as a built-in function. If it is not available, we can easily define it for nonempty lists:

fun reduce(F, [x]) = x

| reduce(F,[x::xs]) = F(x,reduce(F,xs));

Another class of such high order functions is filters that apply a predicate function to elements of a list and return only those elements that satisfy the

1::[2,3] predicate. We can easily write such functions in ML.

A useful functional programming technique is to partially evaluate a function by binding some of its arguments. The result is still a function that may be applied to the remaining arguments. A function some of whose arguments have been bound is called a *closure*. As an example, consider a function Trans-lateWord that takes two arguments: a dictionary to use for translation and the word to translate. The function looks up the word in the dictionary and returns the translation found in the dictionary. We might define closures of this func-tion by binding the dictionary argument to different language dictionaries and producing special translator functions such as ItalianEnglish, ItalianGerman, and EnglishGerman. These new functions are single-argument functions because the dictionary argument has already been bound.

Curried functions may also be used in ML. For example, we can define the function to multiply two integers in curried form:

fun times (x:int) (y:int) = x\*y;

The signature of this function is fn: int-> (int->int). We can build a new function, say multby5, by binding one of the arguments of times:

fun multby5(x) = times(5)(x);

*5.4.1.3 List structure and operations*

The *list* is the major data structuring mechanism of ML; it is used to build a finite sequence of values of the same type. Square brackets are used to build lists: [2, 3, 4], ["a", "b", "c"], [true, false]. The empty list is shown as [] or nil. A list has a recursive structure: it is either nil or it consists of an element followed by another list. The first element of a nonempty list is known as its *head* and the rest is known as its *tail*.

There are many built-in list operators. The two operators hd and tl return the head and tail of a list, respectively. So: hd([1,2,3]) is 1 and tl([1,2,3]) is [2,3]. Of course, hd and tl are polymorphic. The construction operator :: takes a value

and a list of the same type of values and returns a new list: returns

[1,2,3]. We can combine two lists by concatenation: [1,2]@[3] is [1,2,3].

Let us look at some functions that work with lists. First, recall from Chapter 4 the function to reverse a list:



int list

fun reverse(L) = reverse([]) = []

| reverse(x::xs) = reverse(xs) @ [x]

Let us write a function to sort a list of integers using insertion sort:

fun sort(L) = sort([]) = []

| sort(x::xs) = insert (x,xs)

fun insert(x, L) = insert (x,[]) = [x]

| insert (x:int, y::ys) =

if x < y then x::y::ys

else y::insert(x,ys);

The recursive structure of lists makes them suitable for manipulation by recursive functions. For this reason, functional languages usually use lists or other recursive structures as a basic data structuring mechanism in the language.

*5.4.1.4 Type system*

Unlike LISP and APL, ML adopts a strong type system. Indeed, it has an innovative and interesting type system. It starts with a conventional set of built-in primitive types: bool, int, real, and string. Strings are finite sequences of characters. There is a special type called unit which has a single value denoted as (). It can be used to indicate the type of a function that takes no arguments.

As we have seen before, a function has a—po ssibly polymorphic— signature, which is the type of the function. For example, the built-in predicate null which determines whether its argument is the empty list is of type Null is a polymorphic function.

In addition to the built-in type constructors, the programmer may define new type constructors, that is, define new types. There are three ways to do this: type abbreviation, datatype definition, and abstract data type definition. The simplest way to define a new type is to bind a type name to a type expression. This is simply an abbreviation mechanism to be able to use a name rather than the type expression. Some examples are:

type intpair = int \* int;

type ’a pair = ’a \* ’a;

type boolpair = bool pair;

In the second line, we have defined a new polymorphic type called pair which is based on a type ’a. The type pair forms a Cartesian product of two values of type ’a. We have used this type in the third line to define a new monomorphic type.

The second way to define a new type is to specify how to construct values of the new type. For example, similar to Pascal enumeration types, we can define the new type color as:

datatype color = red | white | blue;

**5.4.2 LISP**

The original LISP introduced by John McCarthy in 1960, known as pure LISP, is a completely functional language. It introduced many new program-ming language concepts, including the uniform treatment of programs as data, conditional expressions, garbage collection, and interactive program execution. LISP used both dynamic typing and dynamic scoping. Later ver-sions of LISP, including Scheme, have decided in favor of static scoping. Common Lisp is an attempt to merge the many different dialects of LISP into a single language. In this section, we take a brief look at the LISP family of languages.

*5.4.2.1 Data objects*

LISP was invented for artificial intelligence applications. It is referred to as a language for symbolic processing. It deals with symbols. Values are repre-sented by symbolic expressions (called *S-expressions*). An expression is either an *atom* or a *list*. An atom is a string of characters (letters, digits, and others). The following are atoms:

A

AUSTRIA

68000

A list is a sequence of atoms or lists, separated by space and bracketed by parentheses. The following are lists:

(FOOD VEGETABLES DRINKS)

((MEAT CHICKEN) (BROCCOLI POTATOES TOMATOES) WATER)

(UNC TRW SYNAPSE RIDGE HP TUV)

The empty list “ ()” , also called NIL. The truth value false is represented as () and true as T. The list is the only mechanism for structuring and encoding information in pure LISP. Other dialects have introduced most standard data structuring mechanisms such as arrays and records.

A symbol (as atom) is either a number or a name. A number represents a value directly. A name represents a value bound to the name.

*5.4.2.2 Functions*

There are very few primitive functions provided in pure LISP. Existing LISP systems provide many functions in libraries. It is not unusual Such libraries may contain as many as 1000 functions.

QUOTE is the identity function. It returns its (single) argument as its value. This function is needed because a name represents a value stored in a loca-tion. To refer to the value, we use the name itself; to refer to the name, we use the identity function. Many versions of LISP use 'A instead of the verbose QUOTE A. We will follow this scheme.

The QUOTE function allows its argument to be treated as a constant. Thus, 'A in LISP is analogous to "A" in conventional languages.

Examples

(QUOTE A) = 'A = A

(QUOTE (A B C)) = '(A B C) = (A B C)

There are several useful functions for list manipulations: CAR and CDR are selection operations, and CONS is a structuring operation. CAR returns the first element of a list (like hd in ML); CDR returns a list containing all elements of a list except the first (like tl in ML); CONS adds an element as the first element of a list (like :: in ML). For example

(CAR '(A B C)) = A

The argument needs to be “ quoted,” because the rule in LISP is that a func-tion is applied to the *value* of its arguments. In our case the evaluation of the argument yields the list (A B C), which is operated on by CAR. If QUOTE were missing, an attempt would be made to evaluate (A B C), which would result in using A as a function operating on arguments B and C. If A is not a previously defined function, this would result in an error.

*5.4.2.3 Functional forms*

Function composition was the only technique for combining functions pro-vided by original LISP. For example, the “ to\_the\_fourth” function of Section 5.2 can be defined in LISP as

(LAMBDA(X) (SQUARE (SQUARE X)))

(We assume SQUARE has been defined.) All current LISP systems, however, offer a functional form, called MAPCAR, which supports the application of a function to every element of a list. For example

(MAPCAR TOTHEFOURTH L)

raises every element of the list L to the fourth power.

Rather than provide many functional forms, the choice in LISP has been to supply a large number of primitive functions in the library.

*5.4.2.4 LISP semantics*

One of the most remarkable points about LISP is the simplicity and elegance of its semantics. In less than one page, McCarthy was able to describe the entire semantics of LISP by giving an interpreter for LISP written in LISP itself. The interpreter is called eval.

**5.4.3 APL**

APL was designed by Kenneth Iverson at Harvard University during the late 1950s and early 1960s. Even though APL relies heavily on the assignment operation, its expressions are highly applicative. We will only look at these features here to see the use of functional features in a statement-oriented lan-guage.

*5.4.3.1 Objects*

The objects supported by APL are scalars, which can be numeric or character, and arrays of any dimension. An array is written as sequence of space-sepa-rated elements of the array. Numeric 0 and 1 may be interpreted as boolean values. APL provides a rich set of functions and a few higher-order functions for defining new functions.

The assignment operation (←) is used to bind values to variables. On assign-ment, the variable takes on the type of the value being assigned to it. For example, in the following, the variable X takes on an integer, a character, and an array, in successive statements:

X ← 123;

X ← 'b';

X ← 5 6 7 8 9;

The assignment is an operation that produces a value. Therefore, as in C, it may be used in expressions:

X ← (Y ← 5 6 7 8 9) × (Z ← 9 9 7 6 5);

W← Y - Z;

will set the value of Y to 5 6 7 8 9, Z to 9 9 7 6 5, and W to -4 -3 0 2 4.

*5.4.3.2 Functions*

In contrast to pure LISP, APL provides a large number of primitive functions (called *operations* in APL terminology). An operation is either monadic (tak-ing one parameter) or dyadic (taking two parameters).

All operations that are applicable to scalars also distribute over arrays. Thus, A x B results in multiplying A and B. If A and B are both scalars, then the result is a scalar. If they are both arrays and of the same size, it is element-by-ele-ment multiplication. If one is a scalar and the other an array, the result is the multiplication of every element of the array by the scalar. Anything else is undefined.

The usual arithmetic operations, +, -, ´, ¸, | (residue), and the usual boolean and relational operation,€ Ù, Ú, ~, <, £, =, >, ³, ¹, are provided. APL uses a number of arithmetic symbols and requires a special keyboard.

**5.5 Functional programming in C++**

In this chapter, we have studied the style of functional programming as sup-ported by languages designed to support this style of programming. It is inter-esting to ask to what degree traditional programming languages can support functional programming techniques. It turns out that the combination of classes, operator overloading, and templates in C++ provides a surprisingly powerful and flexible support for programming with functions. In this sec-tion, we explore these issues.

**5.5.1 Functions as objects**

A C++ class encapsulates an object with a set of operations. We may even overload existing operators to support the newly defined object. One of the operator we can overload is the application operator, i.e. parentheses. This can be done by a function definition of the form: operator()(parameters...){...}. We can use this facility to define an object that may be applied, that is, an object that behaves like a function. The class Translate whose outline is shown in Fig-ure 88 is such an object. We call such objects function or functional object. They are defined as objects but they behave as functions.

...definitions of types word and dictionary

class Translate {

private: ...;

public:

word operator()(dictionary& dict, word w)

{

* look up word w in dictionary dict
* and return result

}

}

**FIGURE 88.**Outline of a function object in C++

We may declare and use the object Translate in this way:

Translate Translator(); //construct a Translate object

cout << Translate(EnglishGermanDict, “ university” );

which would presumably print “ universitaet” , if the dictionary is correct.

The ability to define such objects means that we have already achieved the major element of functional programming: we can construct values of type function in such a way that we can assign them to variables, pass them as arguments, and return them as result.

**5.5.2 Functional forms**

Another major element of functional programming is the ability to define functions by composing other functions. The use of such high-order functions is severely limited in conventional languages and they are indeed one of the distinguishing characteristics of functional languages.

**5.5.3 Type inference**

The template facility of C++ provides a surprising amount of type inference. For example, consider the polymorphic max function given in Figure 90. First, the type of the arguments is simply stated to be of some class T. The

template <class T>

T max (T x, T y)

{if (x>y) return x;

else return y;

}

**FIGURE 90.**A C++ generic max function

C++ compiler accepts such a definition as a polymorphic function parameter-ized by type T. We have seen that the ML type inferencing scheme rejects such a function because it cannot infer the type of the operator > used in the function definition. It forces the programmer to state whether T is int or float. C++, on the other hand, postpones the type inferencing to template instantia-tion time. Only when max is applied, for example in an expression ...max(a, b), does C++ do the required type checking. This scheme allows C++ to accept such highly generic functions and still do static type checking. At function definition time, C++ notes the fact that the function is parametric based on type T which requires an operation > and assignment (to be able to be passed and returned as arguments). At instantiation time, it checks that the actual parameters satisfy the type requirements.

We have already contrasted the C++ polymorphic functions with those of ML in terms of type inference. It is also instructive to compare them with those of Ada. In the definition of a polymorphic function based on a type parameter T, neither C++, nor ML require the programmer to state the requirements on type T explicitly: they infer them from the text of the function definition. For example, both discover that type must support the > operation. ML rejects the function definition because of this requirement and C++ accepts it. In Ada, in contrast, the specification of the function must state explicitly that they type T must support the operation >. This is intended to allow the function specifica-tion to be compiled without the body of the function. Both ML and Ada accord special treatment to the assignment and equality operators: Ada refers to types that support these two operations as **private** and ML infers a type that uses the equality operator as not just any type but an *equality* type. Each language tries with its decisions to balance the inter-related requirements of strong typing, ability to describe highly generic functions, writability and readability.

**Logic and rule-based languages 8**

This chapter presents a nonconventional class of languages: logic and rule-based languages. Such languages are different from procedural and functional languages not only in their conceptual foundations, but also in the program-ming style (or paradigm) they support. Programmers are more involved in describing the problem in a declarative fashion, then in defining details of algorithms to provide a solution. Thus, programs are more similar to specifi-cations than to implementations in any conventional programming language. It is not surprising, as a consequence, that such languages are more demanding of computational resources than conventional languages.

**6.1 The"what" versus "how" dilemma: specification versus implementation**

A software development process can be viewed abstractly as a sequence of phases through which system descriptions progressively become more and more detailed. Starting from a software requirements specification, which emphasizes *what* the the system is supposed to do, the description is progres-sively refined into a procedural and executable description, which describes *how* the problem actually is solved mechanically. Intermediate steps are oftenstandardized within software development organizations, and suitable nota-tions are used to describe their outcomes (software artifacts). Typically, a design phase is specified to occur after requirements specification and before implementation, and suitable software design notations are provided to document the resulting software architecture. Thus the "what" stated in the requirements is transformed into the "how" stated in the design document, i.e., the design specification can be viewed as an abstract implementation of the requirements specification. In turn, this can be viewed as the specification for the subsequent implementation step, which takes the design specification and turns it into a running program.

In their evolution, programming languages have become increasingly higher level. For example, a language like Ada, Eiffel, and C++ can be used in the design stage as a design specification language to describe the modular struc-ture of the software and module interfaces in a precise and unambiguous way, even though the internals of the module (i.e., private data structures and algo-rithms) are yet to be defined. Such languages, in fact, allow the module spec-ification (its interface) to be given and even compiled separately from the module implementation. The specification describes "what" the module does by describing the resources that it makes visible externally to other modules; the implementation describes "how" the internally declared data strucures and algorithms accomplish the specified tasks.

All of the stated steps of the process that lead from the initial requirements specification down to an implementation can be guided by suitable systematic methods. They cannot be done automatically, however: they require engi-neering skills and creativity by the programmer, whose responsibility is to map– translate– requirements into executable (usually, procedural) descrip-tions. This mapping process is time-consuming, expensive, and error-prone activities.

An obvious attempt to solve the above problem is to investigate the possibil-ity of making specifications directly executable, thus avoiding the translation step from the specification into the implementation. Logic programming tries to do exactly that. In its simplest (and ideal) terms, we can describe logic pro-gramming in the following way: A programmer simply declares the proper-ties that describe the problem to be solved. The problem description is used by the system to solve the problem (*infer a solution*). To denote its distinctive capabilities, the run-time machine that can execute a logic language is often called an *inference engine*.

In logic programming, problem descriptions are given in a logical formalism, based on first-order predicate calculus. The theories that can be used to describe and analyze logic languages formally are thus naturally rooted into mathematical logic. Our presentation, however, will avoid delving into deep mathematical concepts, and will mostly remain at the same level in which more conventional languages were studied.

**6.1.1 A first example**

In order to distinguish between specification and implementation, and to introduce logic programming, let us specify the effect of searching for an ele-ment x in a list L of elements. We introduce a predicate is\_in (x, L) which is true whenever x is in the list L. The predicate is described using a self-explaining hypothetical logic language, where operator "• " denotes the concatenation of two lists and operator [ ] transforms an element into a list containing it and "iff" is the conventional abbreviation for "if and only if.".

for all elements x and lists L: is\_in (x, L) iff

L = [x]

or

L = L1 • L2 and

(is\_in (x, L1) or is\_in (x, L2))

The above specification describes a binary search in a declarative fashion. The element is in the list if the list consists exactly of that element. Otherwise, we can consider the list as decomposed into a left sublist and a right sublist, whose concatenation yields the original list. The element is in the list, if it is in either sublist.

**6.2 Principles of logic programming**

To understand exactly how logic programs can be formulated and how they can be executed, we need to define a possible reference syntax, and then base on it a precise specification of semantics. This would allow some of the con-cepts we used informally in Section 6.1 (such as "procedural interpretation") to be stated rigorously. This is the intended purpose of this section. Specifi-cally, Section 6.2.1 provides the necessay background definitions and proper-ties that are needed to understand how an interpreter of logic programs works. The interpreter provides a rigorous definition the program’s "procedural interpretation". This is analogous to SIMPLESEM for imperative programs.

**6.2.1 Preliminaries: facts, rules, queries, and deductions**

Although there are many syntactic ways of using logic for problem descrip-tions, the field of logic programming has converged on PROLOG, which is based on a simple subset of the language of first-order logic. Hereafter we will gradually introduce the notation used by PROLOG.

The basic syntactic constituent of a PROLOG program is a *term*. A term is a constant, a variable, or a compound term. A compound term is written as a *functor symbol* followed by one or more arguments, which are themselvesterms. A *ground term* is a term that does not contain variables. Constants are written as lower-case letter strings, representing atomic objects, or strings of digits (representing numbers). Variables are written as strings starting with an upper-case letter. Functor symbols are written as lower-case letter strings.

alpha

125

X

abs (-10, 10)

abs (suc (X), 5)

--this is a constant

--this is a constant

--this is a variable

--this is a ground compound term; abs is a functor

--this is a (nonground) compound term

**6.4 Functional programming versus logic programming**

The most striking difference between functional and logic programming is that programs in a pure functional programming language define functions, whereas in pure logic programming they define relations. In a sense, logic programming generalizes the approach taken by relational databases and their languages, like SQL. For example, consider the simple PROLOG program shown in Figure 99, consisting of a sequence of facts. Indeed, a program of this kind can be viewed as defining a relational table; in the example, a mini-database of classical music composers, which lists the composer’s name, year of birth, and year of death. (See the sidebar on relational database languages and their relation to logic languages.)

In a function there is a clear distinction between the domain and the range. Executing a program consists of providing a value in the domain, whose cor-responding value in the range is then evaluated. In a relation, there is no pre-defined notion of which is the input domain. In fact, all of these possible queries can be submitted for the program of Figure 99:

?- composer (mozart, 1756, 2001).

?- composer (mozart, X, Y).

?- composer X, Y, 1901).

?- composer (X, Y, Z).

In the first case, a complete tuple is provided, and a check is performed that the tuple exists in the database. In the second case, the name of the composer is provided as the input information, and the birth and death years are evalu-ated by the program. In the second case, we only provide the year of death, and ask the program to evaluate the name and year of birth of a composer whose year of death is given as input value. In the fourth case, we ask the sys-tem to provide the name, year of birth, and year of death of a composer listed



**6.5 Rule-based languages**

Rule-based languages are common tools for developing expert systems. Intu-itively, an *expert system* is a program that behaves like an expert in some restricted application domain. Such a program is usually structured as a *knowledge base (KB)*, which comprises the knowledge that is specific to theapplication domain, and an *inference engine*. Given the description of the *current situation (CS)*, often called the database, expressed as a set of facts,the inference engine tries to match CS against the knowledge base to find the rules that can be *fired* to derive new information to be stored in the database, or to perform some action.

An important class of expert system languages (called *rule-based languages*, or *production systems*) uses the so-called *production rules*. Production rules

are syntactically similar to PROLOG rules. Typical forms are:

if condition then action

For example, the MYCIN system for medical consultation allows rules of this kind to be written:

if

description of symptom 1, and

description of symptom 2, and

. . .

description of symptom n

then

there is suggestive evidence (0.7) that the identity of the bacterium is . . .

The example shows that one can state the "degree of certainty" of the conclu-sion of a rule. In general, the action part of a production rule can express any action that can be described in the language, such as updating CS or sending messages.

Supposing that knowledge is represented using production rules, it is neces-sary to provide a reasoning procedure (inference engine) that can draw con-clusions from the knowledge base and from a set of facts that represent the current situation. For production rules there are two basic ways of reasoning:

– *forward chaining*, and

– *backward chaining*.

Different rule-based languages provide either one of these methods or both.

Consider the following initial CS: the alarm is on and switches 1 and 3 are on. The inference engine should help the supervisory system determine the danger level. Forward chaining matches CS against KB, starting from leaf nodes of the and-or tree, and draws conclusions. New facts that are asserted by the rules are added to CS as the rules are fired. In our case, both problems

1 and 2 are asserted, and is subsequently notified, since both

problems 1 and 2 have been discovered.

Suppose now that the purpose of the reasoning procedure was to understand if we are in level 1 of danger. Forward chaining worked fine in the example, since the deduction succeeded. But in general, for a large KB, the same facts might be used to make lots of deductions that have nothing to do with the goal we wish to check. Thus, if we are interested in a specific possible conclusion, forward chaining can waste processing time. In such a case, backward chain-ing can be more convenient. Backward chaining consists of starting from the hypothesized conclusion we would like to prove (i.e., a root node of the and-or tree) and only executing the rules that are relevant to establishing it. In the example, the inference engine would try to identify if problems 1 and 2 are true, since these would cause danger\_level\_1. On the other hand, there is no need to check if danger\_level\_0 is true, since it does not affect danger\_level\_1.To signal problem 1, switches 1 and 3 must be on. To signal problem 2, either the alarm is on or the light is red. Since these are ensured by the facts in CS, we can infer both problems 1 and 2, and therefore the truth of danger\_level\_1.

Different expert system languages based on production rules are commer-cially available, such as OPS5 and KEE. It is also possible to implement production rules and different reasoning methods in other languages; e.g., in a procedural language like C++ or in a functional language like LISP. An implementation in PROLOG can be rather straightforward.

The main difference between logic and rule-based languages is that logic lan-guages are firmly based on the formal foundations of mathematical logic, while rule-based languages are not. Although they have a similar external appearance, being based on rules of the form "if *condition* then *action*", in most cases rule-based languages allow any kind of state-changing actions to be specified to occur in the *action* part.