8 PROCEDURES AND CONCURRENCY: ADA

8.1 DESIGN: CONTROL STRUCTURES

Ada Has a Rich Set of Control Structures

Ada, like most fourth-generation languages, has a richer set of control structures than third-generation languages such as Pascal. These include

1. A conditional
2. An iterator (definite and indefinite)
3. A case-statement
4. Subprograms
5. A goto-statement
6. Facilities for handling exceptions
7. Facilities for concurrent programming

The last two have no corresponding Pascal constructs; the others are generalizations of Pascal.

There Is One, All-Purpose Iterator

Ada provides one iteration statement, the `loop`, which handles definite, indefinite, and even infinite iteration. Although it is quite general and solves some of the problems of previous iterators, it is also quite baroque. The basic loop-statement has the syntax

`loop` ⟨sequence of statements⟩ `end loop`

This is an example of an *infinite* iterator; the sequence of statements is executed forever.

Unending loops are not often needed in programming, so Ada provides an *exit-*

statement for terminating a loop. Executing an exit-statement anywhere within the body of the loop will cause the loop to be aborted and control to be resumed after the `end loop`. For example,[1] this code searches an array `A` for a key `K`:

```
I := A'First;
loop
   if I > A'Last then exit; end if;
   if A(I) = K then exit; end if;
   I := I + 1;
end loop;
```

Notice that there can be any number of exits and that they are embedded at any depth within the loop. Thus, Ada's loop-statement provides mid-decision loops and, in fact, multiple mid-decision loops. Furthermore, by using labels, an exit-statement can exit through any number of levels of nested loops and can even exit from blocks (although not from subprograms). Thus `exit` has some of the characteristics of a nonlocal **goto**.

Since `exit`-statements are so often the consequents of `if`-statements, Ada provides a special abbreviation for this case. Using it, the above loop can be written as follows:

```
I := A'First;
loop
   exit when I > A'Last;
   exit when A(I) = K;
   I := I + 1;
end loop;
```

The point of this abbreviation is not the (approximately 10) characters it saves us from typing; rather, it was the designers' goal that loop termination conditions be clearly marked. Unfortunately, the `exit` can be buried quite deeply in the body of the loop and therefore be hard to spot.

Ada carries this process a step further; since `exit whens` so often appear at the beginning of loops, it permits these loops to be written in a form resembling Pascal's **while**-loops:

```
I := A'First;
while I <= A'Last
loop
   exit when A(I) = K;
   I := I + 1;
end loop;
```

Notice that the `while`-phrase is just a prefix on the basic loop.

Ada provides another abbreviation for cases of definite iteration like that above. This is

[1] `A'First` and `A'Last` are automatically bound to the lower and upper index values of the array `A`.

accomplished by preceding the `loop`-statement with a `for`-phrase instead of the `while`-phrase:

```
for I in A'Range
loop
   exit when A(I) = K;
end loop;
```

Unfortunately, this does not work exactly like the `while`-loop since the `for`-phrase automatically declares the controlled variable `I`, thereby making it local to the loop. This means that outside of the loop it will not be possible to determine where `K` was found. To correct this problem, we must use a different variable as the controlled variable so that we can save `K`'s location in `I`:

```
for J in A'Range
loop
   if A(J) = K then
      I := J;
      exit;
   end if;
end loop;
```

Notice that we are back to using a simple `exit`-statement inside an `if`-statement.

Finally, observe that the above code does not allow us to determine whether `K` was actually found or not, because the control flow reaches the `end loop` in both cases. To solve this problem, we must introduce another variable, as is shown in Figure 8.1.

Figure 8.1 Example of Ada Loop

```
declare
   Found: Boolean := False;
begin
   for J in A'Range
   loop
      if A(J) = K then
         I := J;
         Found := True;
         exit;
      end if;
   end loop;

   if Found then
      ⋮    --Handle found case
   else
      ⋮    --Handle not found case
   end if;
end;
```

■ ***Exercise 8-1*:*** Discuss Ada's `loop`-statement. Are the various abbreviations and special cases justified? Are Ada's loops potentially too unstructured? Design a better loop syntax and defend it.

■ ***Exercise 8-2*:*** Given that Ada has a `goto`-statement, discuss the value of also including an `exit`-statement. Conversely, discuss whether a `goto`-statement should have been included given that Ada has an `exit`-statement.

The Exception Mechanism Is Elaborate

Recall that Ada is intended for embedded computer applications. Since Ada programs may be embedded in devices (such as aircraft) that must act reasonably under a wide variety of conditions and in the face of failures, it is important that Ada programs be able to respond to exceptional situations. This need is satisfied by Ada's *exception mechanism*, which allows us to define exceptional situations, signal their occurrence, and respond to their occurrence.

For example, suppose that a subprogram in an Ada program, `Producer`, is responsible for gathering data and storing it in a stack for later processing by another subprogram, `Consumer`. If the data arrive more rapidly than expected, then `Producer` may attempt to put more data in the stack than will fit. This is an exceptional condition, and it is important that the program handle it appropriately.

Look again at the specification and implementation of the `Stack1` package in Chapter 7, Section 7.4; notice that we have specified an exception, `Stack_Error`, and that the `Push` procedure *raises* this exception if we attempt to push onto a full stack. Thus, we have defined a possible exceptional situation (a mistake in using `Stack1`) and have included code to signal the occurrence of this situation. It remains to define a method of handling this exceptional situation.

In this example, we will suppose that recent data are more important than older data so the proper way to handle stack overflow is to pop a few elements from the stack and then add the new data. This can be accomplished by defining an *exception handler*; see Figure 8.2. If `Push` raises `Stack_Error`, then control proceeds directly to the exception handler for `Stack_Error`, which is at the end of the `Producer` procedure. The execution of `Push` and of the body of `Producer` is aborted.

Let's look at this mechanism in more detail. Whenever we see a construct for binding names, we should ask ourselves, "What is the scope of these names?" We can see that if exceptions had the same Algol-like scope rule as other identifiers in Ada, then the definition of `Stack_Error` in `Producer` would not be visible in `Push`. Thus, exceptions must follow their own rules.

The scope rule that exceptions do follow will be clearer when we discuss the *propagation* of exceptions. Suppose that `Producer` had not defined a handler for `Stack_Error`. What would happen if `Push` raised `Stack_Error`? Ada says that since `Producer` does not provide a handler for `Stack_Error`, the exception will be *propagated* to the caller of `Producer`. If the caller defines a handler for `Stack_Error`, then it will be handled there; otherwise the exception will be propagated to the caller's caller, and so forth. This propagation will continue until a handler is found or until the outermost procedure is reached, in which case the program will be terminated.

Now we can see the essence of the scope rule for exceptions. If the exception is defined

```
use Stack1;
procedure Producer (...);
begin
    .
    .
    .
  Push (Data);
    .
    .
    .
exception
  when Stack_Error =>
    declare Scratch: Integer;
    begin
      for I in 1..3 loop
        if not Empty then Pop(Scratch); end if;
      end loop;
      Push(Data);
    end;
end Producer;
```

Figure 8.2 Definition of an Exception Handler

in the local environment, we go to its handler; otherwise we look for a handler in the caller's environment. We continue down the dynamic chain, going from each subprogram to its caller, until we find an environment that defines a handler for the exception. Thus, with regard to exceptions, subprograms are *called in the environment of the caller*, although in all other respects they are called in the environment of definition. Although all other names in Ada are bound statically, exceptions are bound dynamically. This makes exceptions something of an exception themselves! It also makes Ada more complex since it violates both the Structure and Regularity Principles.

Exceptions must be implemented almost exactly the way we have described them above. When an exception is raised, a run-time routine must scan down the dynamic chain looking for an environment containing a handler for the exception. During this scan the activation record of any subprogram or block that does not define a handler for the exception must be deleted. Thus we can see that an exception is something like a dynamically scoped nonlocal **goto**.

■ ***Exercise 8-3:*** Explain why Ada's exceptions violate the Structure Principle.

■ ***Exercise 8-4**:*** Define and describe a *statically scoped* exception mechanism for Ada. How is it less general than Ada's mechanism?

■ ***Exercise 8-5*:*** Evaluate the following argument for a dynamically scoped exception mechanism: A dynamic scope rule is the only useful scope rule for an exception mechanism. It is often the case that different callers of a subprogram will want to handle exceptions arising from that subprogram in different ways. If the handler for an exception were bound in a subprogram's environment of definition, it would be fixed for all time. In that case, it might as well be made part of the subprogram.

Parameters Can Be In, Out, or In Out

In previous chapters, we have seen a number of different *modes* for passing parameters to subprograms, namely, value, reference, name, constant, and value-result. All of these techniques have some advantages and some disadvantages; none of the languages we have studied seems to provide just the right set of modes. Ada, however, provides three parameter passing modes that seem to come close to achieving this. We will discuss why later; first, we will investigate these modes.

Ada's parameter passing modes reflect the ways in which the programmer may intend to use the parameter: input, output, or both input and output. These modes are indicated by the reserved words `in`, `out`, or `in out` preceding the type of the formal parameter in the formal parameter list. If the mode is omitted, then `in` is assumed.

`In` parameters are used to transmit information into a procedure but not out of it; they are essentially the same as the *constant* parameters described in the original Pascal Report (see Chapter 5, Section 5.5). Since these parameters are for input, the language allows only read access to them; assignments to formal `in` parameters are illegal (i.e., `in` parameters act like constants within the body of the procedure).

Recall that pass as constant leaves it up to the compiler to determine whether a parameter's value or address is actually passed. Ada does not go quite this far; it specifies that parameters of *elementary types* and pointers will always be copied (i.e., passed by value). For certain other types, including *composite types* (e.g., arrays and records), the compiler may choose either to copy the value (if it is short) or to pass its address. For the remaining types the parameter is always passed by reference.[2]

`Out` parameters are just the opposite of `in`; they are used for transmitting results out of a subprogram. In Ada 83 these parameters were considered write-only; within the subprogram they could be used as a destination but not as a source. This was found to be inconvenient, so Ada 95 allows an `out` parameter to be read within the subprogram (after, presumably, it has been stored into). Obviously, the actual parameter corresponding to an `out` formal must be something into which it is meaningful to store, that is, a variable of some type.

`Out` parameters are quite efficient: Just as for `in` parameters, elementary `out` parameters are copied out, and most composite parameters may be either passed by reference or copied out. Note that copying out the value of an `out` parameter is essentially the result half of pass by value-result (Chapter 2, Section 2.3).

The remaining parameter passing mode is `in out`; this is used for parameters that are to be used as both a source and a destination by the subprogram. Since the actual parameter is potentially a destination, it must be a variable of some sort, just as for `out` parameters.

The same implementation methods are used for `in out` parameters: For elementary values the values are copied in on call and out on return, which is essentially pass by value-result. Most composite parameters may be passed by reference or value-result, compiler's choice.

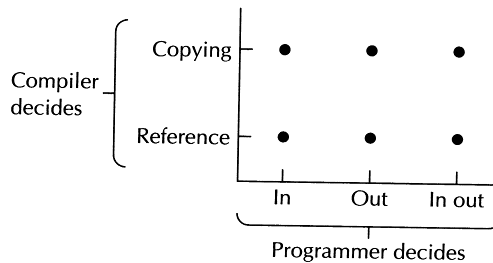
Ada seems to have solved the problems that we saw in the parameter passing modes of FORTRAN, Algol-60, and Pascal. The reason is that Ada's solution is more *orthogonal*, that

[2] The precise distinction between by-copy, by-reference, and other parameters is too complicated to reproduce here.

is, it better separates the independent functions. In the case of parameters, there are two issues:

1. How the parameter is to be used (i.e., input, output, or both input and output)
2. How the parameter transmission is to be implemented (i.e., pass by reference, or pass by value and/or result)

The first issue is a *logical* issue, that is, it affects the input-output behavior of the program. The second issue is a *performance* issue, that is, it affects the efficiency of the program. Ada allows the programmer to resolve the first issue (by specifying a parameter as `in`, `out`, or `in out`). It reserves to the compiler the right to resolve the second issue since requiring the programmer to make this choice would introduce a machine dependency into the programs (and violate the Portability Principle). The other languages we have discussed garbled these two issues; Ada's orthogonal solution can be visualized as follows:



Ada slightly mixes two other issues that should be orthogonal, namely, the reference/copying issue and the parameter's type, since it specifies that elementary parameters are always copied.

How does the compiler decide whether to pass a parameter by reference or by copying? This is easy to analyze. Suppose that a composite parameter occupies s words (or whatever the units of storage may be) and that each component of the parameter occupies one word. This would be the case if, for example, the parameter were an array of integers. Further, suppose that during the execution of the subprogram components of this parameter are accessed n times. We can compute C, the cost of passing the parameter by copying, and R, the cost for reference.

If the parameter is copied, then $2s$ memory references will be required to transmit it and one memory reference will be required for each of the succeeding n accesses to components. Hence,

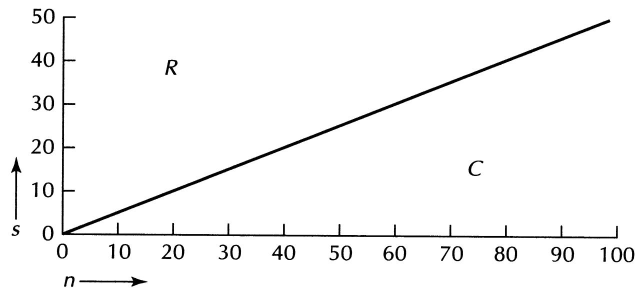
$$C = 2s + n$$

If the parameter is passed by reference, then one memory reference is required to transmit it (assuming an address fits in one storage unit), and two are required for each of the succeeding n accesses to components. The latter is true because a reference parameter must be accessed indirectly.[3] Hence,

$$R = 2n + 1$$

[3] We are ignoring the possibility that the compiler might optimize code by keeping the address of the actual parameter in a register across several accesses.

Now we can determine the conditions under which it is less expensive to pass a parameter by reference than by copying. $R < C$ whenever $2n + 1 < 2s + n$, hence $R < C$ whenever $n < 2s - 1$. We can see the regions where reference or copy is better in the following diagram:



Unfortunately, since n depends on the dynamic behavior of the program, it is usually impossible for a compiler to determine this value. Therefore, the decision of whether to use value or reference parameters must be based on some plausible assumptions. For example, if we assume that the parameter is an array and each array element is referenced once, then $n = s$ (if each element occupies one storage unit), $C = 3n$, and $R < C$; so pass by reference is less expensive. When will copying be less expensive than pass by reference? Note that $R > C$ when $n > 2s - 1$. Therefore, copying is less expensive if on the average each array element is accessed at least twice.

■ ***Exercise 8-6*:*** Is it a good or a bad idea that Ada specifies that elementary parameters always be copied (passed by value and/or result)? Discuss the factors involved, including representation independence, efficiency, and predictability.

■ ***Exercise 8-7:*** Write a subprogram whose results differ depending on whether a given *composite* parameter is passed by reference or by copying. This permits us to determine the parameter passing methods used by a compiler (and also shows that Ada programs may be implementation sensitive).

■ ***Exercise 8-8*:*** A program that depends on the implementation of Ada, such as the one you wrote in the previous exercise, is not considered legal Ada. Unfortunately, it is very difficult for a compiler to check for this situation. Is this a good design decision? Discuss the philosophy of defining errors that cannot be practically checked by a compiler. Is there any alternative? Discuss and evaluate the possibilities.

■ ***Exercise 8-9:*** The above analysis of the pass by reference/pass by copy trade-off applies to `in` and `out` parameters (i.e., parameters that are copied once). Extend this analysis to `in out` parameters, which must be copied twice. Assume that an address takes two units of storage (e.g., two bytes) rather than one.

Position-Independent and Default Parameters

Suppose we are implementing an Ada package to draw graphs. The package might contain this specification of a routine to draw the axes of the graph:

```
procedure Draw_Axes (X_Origin, Y_Origin: Coord;
   X_Scale, Y_Scale: Float; X_Spacing, Y_Spacing: Natural;
   X_Logarithmic, Y_Logarithmic: Boolean;
   X_Labels, Y_Labels: Boolean; Full_Grid: Boolean);
```

Procedures with a large number of parameters, such as this, are not uncommon in subprogram libraries that are trying to achieve generality and flexibility. A call to `Draw_Axes` can look like this:

```
Draw_Axes (500, 500, 1.0, 0.5, 10, 10,
           False, True, True, True, False);
```

It is very difficult to tell the meanings of the parameters from this call; even if programmers know the purpose of `Draw_Axes`, they will have to look up its definition to find out the meaning of the parameters.

The problem with the above procedure specification is that the order of the parameters is essentially arbitrary. Although it makes sense that an *x*-related parameter always precedes the corresponding *y*-related parameter, there is no particular reason why the scale factors should precede the spacings or the requests for labels follow the requests for logarithmic spacing. People remember things most easily when they are related meaningfully. When there are no meaningful relationships, the only alternate is rote memorization, which is error-prone.

Operating system command languages long ago found a solution to the problem of programs with many parameters: *position-independent parameters*. The basic idea is that the parameters can be listed in any order. Then, to determine which is which, a name is associated with each parameter; this name identifies the parameter's function. Our `Draw_Axes` example can be written using position-independent parameters in Ada as follows:

```
Draw_Axes (X_Origin => 500, Y_Origin => 500,
   X_Spacing => 10, Y_Spacing => 10, Full_Grid => False,
   X_Scale => 1.0, Y_Scale => 0.5,
   X_Label => True, Y_Label => True,
   X_Logarithmic => False, Y_Logarithmic => True );
```

This is more readable and much less prone to mistakes than the position-dependent version.

Position-independent parameters are another illustration of the Labeling Principle.

> **The Labeling Principle**
>
> Avoid arbitrary sequences more than a few items long; do not require the programmer to know the absolute position of an item in a list. Instead, associate a meaningful label with each item, and allow the items to occur in any order.

We have already seen several examples of this principle: All programming languages provide symbolic names for variables rather than requiring the programmer to remember the absolute memory location of the variables. We have also seen (Chapter 5, Section 5.5) the advantages of Pascal's *labeled* case-statement over earlier *unlabeled* case-statements.

Ada provides an additional facility, also suggested by operating system control languages, that can improve the readability of our call on `Draw_Axes`; these are *default* para-

meters. The motivation for these is simple: In an attempt to be general, `Draw_Axes` provides many different options. There are several of these that will be rarely used; for example, most users will not want a full grid or logarithmic axes. Unfortunately, all users must specify these options, if only to disable them, in order for them to be available to a small fraction of the users. This is actually a violation of the Localized Cost Principle, which says that users should not have to pay for what they do not use.

Default parameters solve this violation of the Localized Cost Principle by permitting the designer of a subprogram to specify *default values* that are supplied for parameters if the corresponding actuals are omitted. The following is a reasonable set of defaults for `Draw_Axes`:

```
procedure Draw_Axes (X_Origin, Y_Origin: Coord := 0;
   X_Scale, Y_Scale: Real := 1.0;
   X_Spacing, Y_Spacing: Natural := 1;
   X_Label, Y_Label: Boolean := True;
   X_Logarithmic, Y_Logarithmic: Boolean := False;
   Full_Grid: Boolean := False );
```

This declaration permits us to write the preceding call to `Draw_Axes` in a much more compact form:

```
Draw_Axes(500, 500, Y_Scale => 0.5, Y_Logarithmic => True,
   X_Spacing => 10, Y_Spacing => 10);
```

Notice that position-dependent and position-independent parameters can be mixed in a single call; the *x* and *y* origins are specified position dependently in the above call.

Position-Independent and Default Parameters Complicate Operator Identification

Of course, all this flexibility does not come without a cost. A major part of the cost is the increased complexity of the language manual and the greater number of constructs that the programmers must learn. In Ada, however, there is a less obvious cost that results from *feature interaction*, in this case, the interaction of overloading with position-independent and default parameters.

Recall that an overloaded subprogram can have several meanings in one context; the compiler determines the meaning intended on the basis of the types of the parameters and the context of use of the subprogram. Consider the following procedure declarations, both occurring in the same scope:

```
procedure P (X: Integer; Y: Boolean := False);
procedure P (X: Integer; Y: Integer := 0);
```

The procedure `P` is overloaded because it bears two meanings at once. The rules of operator identification tell us that `P(9, True)` is a call on the first procedure and `P(5, 8)` is a call on the second. Notice, however, that we have provided a default value for `Y` in both procedure declarations. What is the meaning of the call `P(3)`? It could be either one since

we have omitted the only parameter that distinguishes the two overloadings. In fact, because the call `P(3)` is ambiguous, Ada does not allow the two procedure declarations shown above. A set of subprogram declarations is illegal if it introduces the *potential* for ambiguous calls.

These potential ambiguities can arise in many ways. Consider the following declarations:

```
type Primary is (Red, Blue, Green);
type Stop_Light is (Red, Yellow, Green);

procedure Switch (Color: Primary; X: Float; Y: Float);
procedure Switch (Light: Stop_Light; Y: Float; X:Float);
```

These look quite different, and there are no defaults. Unfortunately, the call

```
Switch (Red, X => 0.0, Y => 0.0);
```

is ambiguous, so the declarations are illegal. Here we can see the interaction of *two overloadings*—an overloaded enumeration and an overloaded procedure.

The rules that specify what overloadings are allowed are actually quite a bit more complicated than we have described. Suffice it to say that both the human reader and the compiler can have difficulty with a program that makes extensive use of overloading and position-independent and default parameters.

■ ***Exercise 8-10:*** Explain in detail why the preceding call of `Switch` is illegal.

■ ***Exercise 8-11**:*** We have seen the benefits both of overloading and of position-independent and default parameters. We have also seen some of the complications that result from the interaction of these features. Has Ada made the right choice? Either propose your own alternative and argue that it is better than Ada's or show that Ada's solution is better than the alternatives.

Ada Permits Concurrent Execution

When people set out to accomplish some task, it is often more efficient and more convenient to do several things at the same time. For example, one person may read a map while the other drives. There would not be much point in finishing the driving before the map reading is started, and it would not be very efficient to do all of the map reading before starting the driving. The same holds true in programming; therefore, Ada provides a *tasking* facility that allows a program to do more than one thing at a time.

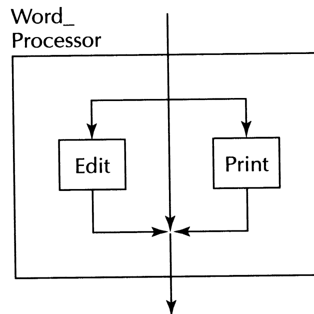
Let's consider an example. Suppose we have a small, stand-alone word-processing system that allows users to print one file while they are editing another. This is programmed in Ada by defining two disjoint tasks, say `Print` and `Edit`; see Figure 8.3. Here we have a procedure, `Word_Processor`, with two local tasks, `Edit` and `Print`. Notice that a task is declared very much like a package, with a separate specification and body. In this case, there are no public names so there is nothing in the specification.

When we call `Word_Processor`, all local tasks are automatically initiated. That is, we do three things at once: We begin executing the bodies of `Word_Processor`, `Edit`, and `Print`. We can assume that `Edit` does its job of communicating with the user and

```
procedure Word_Processor is
   task Edit; end Edit;
   task body Edit is
   begin
      :      --edit the file selected
   end;

   task Print; end Print;

   task body Print is
   begin
      :      --print the file selected
   end;

begin
   --initiate tasks and wait for their completion
end Word_Processor;
```

Figure 8.3 Noncommunicating Tasks

editing the file while `Print` does its job of listing another file on the printer. What does `Word_Processor` do? In this example, not very much. Since the body of the procedure is empty, we immediately encounter the `end` and try to return. Ada, however, prevents a procedure from returning as long as it has active local tasks. Hence, `Word_Processor` will wait as long as `Edit` and `Print` are executing. When they have both finished (by reaching their `end`-statements), then `Word_Processor` will be able to return to its caller. This can be visualized as follows:



Only when `Edit`, `Print`, and the body of `Word_Processor` have all reached their `end`s can `Word_Processor` exit.

Why does Ada require all local tasks to finish before a procedure can exit? Suppose that `Word_Processor` had some local variables; these are visible to `Edit` and `Print` because they are declared in the same environment. Consider what happens when `Word_Processor` exits; like all procedures, its activation record is deleted. Any references

in `Edit` and `Print` to the local variables of `Word_Processor` will now be meaningless *dangling references*. One alternative is to preserve `Word_Processor`'s activation record until `Edit` and `Print` have finished. This in turn precludes using a simple stack discipline for activation record allocation and deallocation.[4] The alternative is to delay `Word_Processor`'s return until it can be done safely; this is an example of the Security Principle.

■ ***Exercise 8-12*:*** Discuss alternative solutions to the dangling reference problem. Either investigate nonstacked activation records, or suggest another approach that prevents dangling references and still permits the use of a stack.

Tasks Synchronize by Rendezvous

We have seen an example of two tasks that execute concurrently, but do not communicate. This is not usually the case; consider our driving and map-reading example. If the concurrent map reading is to be useful, it will be necessary for the map reader to communicate directions to the driver. It may even be necessary for the driver to *synchronize* with the navigator by pulling off the road and waiting for the navigator to decide where they should turn next.

The same is the case in programming. Suppose we have an application that retrieves records from a database, summarizes them, and prints the results. Since the retrieval and summarization processes are fairly independent, we can implement them as tasks that run concurrently. However, they will have to communicate; the `Summary` task will have to tell the `Retrieval` task which record it wants, and the `Retrieval` task will have to transmit the records to the `Summary` process when they are found. An outline of the program appears in Figure 8.4.

Notice that the specification of `Retrieval` contains two `entry` declarations. We can see that these have parameters very much like procedures, and, in fact, they can be called like procedures. For example, the body to `Summary` might look like this:

```
task body Summary is
begin
    .
    .
    .
   Seek (ID);
    .
    .
    .
   Fetch (New_Recd);
    .
    .
    .
end Summary;
```

The `Seek` call tells `Retrieval` to find the record in the database, and the `Fetch` call puts the sought record's value in `New_Recd`. Although procedures and entries are similar in some ways, there are important differences.

---

[4] This would be an example of a *retention* strategy, such as discussed in Chapter 6, Section 6.2.

```
procedure DB_System is
  task Summary; end Summary;
  task body Summary is
  begin
    :     --generate the summary
    .
  end;

  task Retrieval;
    entry Seek (K: Key);
    entry Fetch (R: out Recd);
  end Retrieval;

  task body Retrieval is
  begin
    :     --seek record and return it
    .
  end;

begin
  --await completion of local tasks
end DB_System;
```

Figure 8.4 Communicating Tasks

Recall how a normal procedure call works: When one subprogram calls another, the parameters are transmitted from the caller to the callee, the caller is suspended, and the callee is activated. The caller remains suspended until the callee returns; at that time the results are transmitted from the callee back to the caller, the callee is deactivated, and the caller is resumed (i.e., reactivated).

When one task calls an `entry` in another, it transmits parameters very much like a procedure call. The difference is that once the callee accepts the parameters, the caller continues executing; it is not suspended. The two tasks remain active and continue to execute concurrently. Thus, the call `Seek(ID)` is more properly viewed as a *message-sending* operation in which the message (`ID`) is put into the `entry Seek` where it is available to `Retrieval`. In fact, an `entry` is often called a *mailbox* or *message port.*

How does a task accept a message? This is accomplished with an *accept*-statement, which has the syntax

`accept` ⟨name⟩ `(`⟨formals⟩`)` `do` ⟨statements⟩ `end` ⟨name⟩`;`

For example, the body of `Retrieval` might look like the following:

```
task body Retrieval is
begin
  loop
    accept Seek (K: Key) do
      RK := K;
```

```
      end Seek;
      :     --seek record RK and put in Recd_Value
      .
      accept Fetch (R: out Recd) do
        R := Recd_Value;
      end Fetch;
    end loop;
end Retrieval;
```

Notice that Retrieval is written as a loop that alternatively seeks a record and returns its value. The first accept-statement accepts a message from the Seek mailbox and binds the formal parameter K to this message.

What happens if Retrieval reaches this accept-statement before Summary has sent the message? Does Retrieval come away empty handed? No, just as in our map-reading example, it "pulls over to the side of the road" and waits for a message in Seek. In other words, it suspends itself awaiting the arrival of a message in the mailbox Seek. Then, when Summary sends the message, it will be accepted and both tasks will proceed concurrently. Similarly, if Summary attempts to transmit Seek(ID) before Retrieval is ready to accept it, Summary will be suspended until Retrieval accepts a message from Seek. Thus, a message is only actually transmitted when the entries to which the sender calls and from which the receiver accepts are the same. This meeting is called a "rendezvous" in the Ada literature. The rendezvous may never take place, so one or both tasks may wait forever, a situation called *dead-lock.*

The structure of the database system is pictured in Figure 8.5. The message ports (mailboxes, entries) are shown as circles. You may be surprised to see that the arrow for Fetch goes from Summary to Retrieval. This is because Summary places a message in Fetch to request that the value of the record be put in an output parameter (New_Recd); Retrieval accepts this message and puts the value in New_Recd.

Guards and Protected Types Control Concurrent Access to Data Structures

The Ada 83 task model was found to have some limitations. To understand these, let's extend our stand-alone word-processing example. As it stands, the two tasks do not commu-

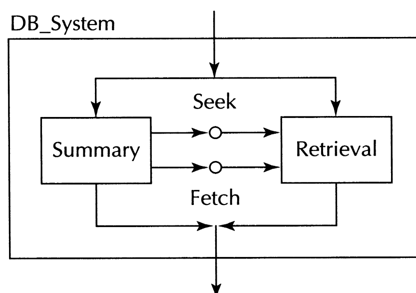


Figure 8.5 Structure of Concurrent Database System

nicate; more realistically we would expect `Edit` to send documents to `Print` for printing, and this could be accomplished by a rendezvous. For example, `Print` could be given an `entry` for the document to be printed and an `entry` to tell it to terminate:

```
task Print is
  entry Send (D: Document);
  entry Terminate;
end Print;
```

The body would then be a loop waiting for documents to print or for a `Terminate` signal:

```
task body Print is
begin
  loop
    select
      accept Send (D: Document) do
      ... print the document ...
      end Send;
      or accept Terminate do exit end Terminate;
    end select;
  end loop;
end Print;
```

The `Edit` task requests the printing of a document by `Print.Send(D)`.

This works, but the tasks are quite *tightly coupled.* Since printing is a comparatively slow operation, we can expect that sometimes the user will ask to print a second document before the printing of the first has been completed. In this case, `Print` will not have returned to the `accept Send`, and so when `Edit` gets to `Print.Send(D)` its execution will be blocked until `Print` has completed the first document.

As you probably know, the system can be made more *loosely coupled* by putting a *buffer* between the two tasks. To accomplish this we can use the `Communication` package described in Section 7.4, which implements a circular buffer. To hide the implementation details two new procedures, `Send` (used by `Edit`) and `Receive` (used by `Print`), are defined (Figure 8.6).

```
package Communication is
  Size: constant Integer := 100;
  Avail: Integer range 0 .. Size := 0;
  procedure Send (D: in Document);
  procedure Receive (D: out Document);
private
  In_Ptr, Out_Ptr: Integer range 0 .. Size-1 := 0;
  Buffer: array (0 .. Size-1) of Document;
end Communication;
```

Figure 8.6 Specification of Communication as a Package

We have also added a variable `Avail` that indicates the number of documents in the buffer, so that we will not `Send` to a full buffer or `Receive` from an empty one. (The `Edit` and `Print` tasks have the responsibility of doing these checks.) A simple implementation of the `Communication` package is shown in Figure 8.7.

Unfortunately, this implementation is incorrect, since the `Print` task might be executing `Receive` at the same time that the `Edit` task is executing `Send`. Thus it is possible for one task to be executing `Avail := Avail - 1` at the same time the other is executing `Avail := Avail + 1`; the result is that the variable may be updated incorrectly.

Exercise 8-13*: Show in detail how `Avail` can be updated incorrectly and explain the consequences. Could it be too large? Too small? Either?

One way to solve this problem is to make `Communication` a task, and make `Receive` and `Send` *guarded entries* (Figure 8.8).

The `accept`-statements have guards `when` ⟨condition⟩ on them so that they will accept a message only if the condition is satisfied. If the producer `Edit` attempts to `Send` to a full buffer, or if the consumer `Print` tries to `Receive` from an empty buffer, the attempt will block, until the other task changes the situation and allows it to proceed. Further, since `Communication` is now an independent task, it can be executing only one of the entries `Send` and `Receive` at a time, and inconsistent updating of `Avail` is not possible. Unfortunately we now have three tasks, `Edit`, `Print`, and `Communication`, which manages the buffer between the other two.

This solution solves the problem, but at the expense of an additional task that, in effect, actively waits for a `Send` or `Receive` request. This is an inefficient way to solve a common problem, controlling concurrent access to a shared data structure, and so Ada 95 has an additional construct, the *protected type*, which accomplishes it more directly; Figure 8.9 shows a definition of `Communication` as a protected type.

```
package body Communication is
   procedure Send (D: in Document);
   begin
      Buffer (In_Ptr) := D;
      In_Ptr := (In_Ptr + 1) mod Size;
      Avail := Avail + 1;
   end Send;
   procedure Receive (D: out Document) is
   begin
      D := Buffer (Out_Ptr);
      Out_Ptr := (Out_Ptr + 1) mod Size;
      Avail := Avail - 1;
   end Receive;
end Communication;
```

Figure 8.7 Implementation of Communication as a Package

```
task Communication is
  entry Send (D: in Document);
  entry Receive (D: out Document);
  entry Terminate;
private
  Size: constant Integer := 100;
  Avail: Integer range 0 .. Size := 0;
  Buffer: array (0 .. Size-1) of Document;
  In_Ptr, Out_Ptr: Integer range 0 .. Size-1 := 0;
end Communication;

task body Communication is
begin
  loop
    select
      when Avail < Size accept Send (D: in Document)
      do
        Buffer (In_Ptr) := D;
        In_Ptr := (In_Ptr + 1) mod Size;
        Avail := Avail + 1;
      end Send;
      or when Avail > 0 accept Receive (D: out Document)
      do
        D := Buffer (Out_Ptr);
        Out_Ptr := (Out_Ptr + 1) mod Size;
        Avail := Avail - 1;
      end Receive;
      or accept Terminate do exit end Terminate;
    end select;
  end loop;
end Communication;
```

Figure 8.8 Communication as a Task

Since this is a type definition, it defines a template, which must be instantiated with an object declaration to create an actual buffer. For example,

```
Channel: Communication;
```

creates an instance of `Communication`. Within the scope of this declaration the `Edit` and `Print` tasks can communicate by means of `Channel.Send(D)` and `Channel.Receive(D)` requests. The effect of `Buffer` belonging to a protected type is that only one of its entries can be executed at a time, which ensures that the buffer will be managed con-

```
protected type Communication is
   entry Send (D: in Document);
   entry Receive (D: out Document);
private
   Size: constant Integer := 100;
   Avail: Integer range 0 .. Size := 0;
   Buffer: array (0 .. Size-1) of Document;
   In_Ptr, Out_Ptr: Integer range 0 .. Size-1 := 0;
end Communication;

protected body Communication is
begin
   entry Send (D: in Document) when Avail < Size is
   begin
      Buffer (In_Ptr) := D;
      In_Ptr := (In_Ptr + 1) mod Size;
      Avail := Avail + 1;
   end Send;

   entry Receive (D: out Document) when Avail > 0 is
   begin
      D := Buffer (Out_Ptr);
      Out_Ptr := (Out_Ptr + 1) mod Size;
      Avail := Avail - 1;
   end Receive;

end Communication;
```

Figure 8.9 Communication as a Protected Type

sistently, but without the expense of a third task. If one of the entries is active, then a call on the other will block until the first is done.

■ ***Exercise 8-14**:*** We have only touched on the essentials of Ada's task facility. Look up concurrency in the Ada Reference Manual and evaluate its other features. Are there too many features? Is the language too complex in this area? Are there too few features? Are there important things that cannot be easily accomplished with Ada's tasks? Explain or defend your answers.

8.2 DESIGN: SYNTACTIC STRUCTURES

Ada Follows Pascal in the Algol Tradition

In most respects Ada's syntactic conventions follow the Pascal tradition (which is in turn in the Algol tradition), although in some cases they have been made more systematic. For ex-

ample, in Chapter 6 we saw that blocks can be considered degenerate procedures. This similarity is reflected in Ada's syntax for blocks and procedures:

```
declare                              procedure ⟨name⟩ (⟨formals⟩) is
   ⟨local declarations⟩                 ⟨local declarations⟩
begin                                begin
   ⟨statements⟩                         ⟨statements⟩
exceptions                           exceptions
   ⟨exception handlers⟩                 ⟨exception handlers⟩
end;                                 end;
```

The difference between procedures and blocks is that procedures have a name and formals (and hence can be called from different contexts with different parameters) whereas blocks do not (since they are "called" where they are written). We have also seen that functions and task bodies follow a similar pattern.

There are other cases where Ada has made syntactic similarities reflect semantic similarities. For example, Ada's notation for a variant record, which defines several different cases for a record, has been made to look like a case-statement. Also, the notation for position-independent actual parameters is similar to the notation for initializing arrays and other composite data structures. These are examples of the Syntactic Consistency Principle.

The Syntactic Consistency Principle

Things that look similar should *be* similar; things that *are* different should look different.

Another difference between Pascal and Ada is the use of semicolons. In Pascal, as in Algol, semicolons are *separators*: they come *between* statements. Thus, there is a similarity between infix operators separating their operands:

$(E_1 + E_2 + E_3)$

and semicolons separating statements:

begin S_1 ; S_2 ; S_3 **end**

This convention creates some minor maintenance problems since if we need to insert a new statement before the **end** in a compound statement like the following:

```
begin
   S1;
   S2;
   S3
end
```

we must remember to add a semicolon to S_3. Fortunately, both Algol and Pascal allow *empty statements*, so we can write

```
begin
   S1;
   S2;
   S3;
end
```

You cannot see it, but there is an empty statement between the last semicolon and the **end**. Many Pascal programmers terminate each statement with a semicolon, which simplifies editing.

Ada has adopted a different convention, a *terminating semicolon*, for the reason described above, and also because some studies suggest that a terminating semicolon is less error-prone. Both statements and declarations in Ada end with semicolons, even before `end`s, and there is no need for an empty statement.

Ada Uses a Fully Bracketed Syntax

Recall that one of the contributions of Algol was the *compound statement*, which allows statements to contain other statements. This led to the ability to structure programs hierarchically and spurred interest in structured programming. The compound statement idea had a flaw, however; look at these Pascal constructs:

```
for i := ... do begin ... end
if ... then begin ... end else begin ... end
procedure ... begin ... end
function ... begin ... end
case ... of a: begin ... end; ... end
while ... do begin ... end
with ... do begin ... end
record ... end
```

Notice that all of these compound structures end with the same keyword, **end**. Therefore, if the programmer omits an **end**, it is very likely that the compiler will discover it only at the end of the program and that all the **begins** and **end**s will be matched up incorrectly. Similarly, it can be quite difficult for a human reader looking at an **end** to tell exactly what it is ending.

One solution to this problem is to use a variety of different kinds of brackets. For instance, in mathematics we have (...), {...}, [...]. About the time Pascal was being designed, language designers were experimenting with this solution to the **begin–end** problem, although the ideas go back to at least Algol-58. The result—a *fully bracketed* syntax—arrived too late for Pascal, but was adopted by Ada 83. This means that each construct has its own kind of brackets; for example, in Ada we have

```
loop ... end loop
if ... end if
case ... end case
record ... end record
```

For constructs that are named, such as subprograms, packages, entries, and tasks, a unique bracket is made by attaching the name to 'end':

```
function ⟨name⟩ (⟨formals⟩) is ... begin ... end ⟨name⟩;
procedure ⟨name⟩ (⟨formals⟩) is ... begin ... end ⟨name⟩;
package ⟨name⟩ is ... end ⟨name⟩;
package body ⟨name⟩ is ... end ⟨name⟩;
accept ⟨name⟩ (⟨formals⟩) do ... end ⟨name⟩;
```

This allows the compiler to do better error checking since it can ensure that an `end` goes with the correct declaration.

The fully bracketed syntax interacts with the `if`-statement in an interesting way: It is common to break the flow of control into $n + 1$ paths $P_1, P_2, ..., P_n, P_{n+1}$ on the basis of conditions $C_1, C_2, ..., C_n$. This leads to a program that looks like the following (where $n = 3$):

```
┌ if C1 then
│     P1
├ else
│  ┌ if C2 then
│  │  P2
│  ├ else
│  │  ┌ if C3 then
│  │  │     P3
│  │  ├ else
│  │  │     P4
│  │  └ end if;
│  └ end if;
└ end if;
```

Although the indenting accurately reflects the way in which the `if`-statements are built up from other `if`-statements, it does not accurately reflect the intentions of the programmer, which is that there are four similar cases. It actually *violates* the Structure Principle. For this reason Ada provided a case-statement-like syntax for expressing these `else-if` chains:

```
┌ if C1 then
│     P1
├ elsif C2 then
│     P2
├ elsif C3 then
│     P3
├ else
│     P4
└ end if;
```

This is an idea that originated in the 1960s in LISP (which is discussed in Chapters 9, 10, and 11). It conforms better to both the Structure Principle and the Syntactic Consistency Principle at the cost of some increased syntactic complexity (thus slightly violating the Simplicity Principle). It is typical of the trade-offs that must be made in language design.

8.3 EVALUATION AND EPILOG

Ada Is an Engineering Trade-Off

Ada is certainly not a perfect language. We have seen that in many cases the designers of Ada had to trade off satisfying one principle for satisfying another. In many cases, the trade-off was implicit: providing some facility versus keeping the language simple. Trade-offs of this sort are common in any kind of engineering design, and different designers will make the trade-offs in different ways, depending on the weight they attach to different factors. For example, adding an air conditioner to a car will decrease its fuel efficiency. Whether this is a good trade-off or not depends a great deal on the climate in which the car will be used, the cost of fuel, and the personal preferences (or values) of the occupants. (Recall the distinction between efficiency and economy discussed in Chapter 4, Section 4.3.)

DoD Has Disallowed Subsets

One solution that has been adopted in many engineering problems is to provide options; this allows customers to make the trade-offs for themselves. For example, air-conditioning on a car is usually an option. Unfortunately, this solution was not permitted for Ada; the Department of Defense said that there would be no Ada subsets or supersets. In other words, there would be only one Ada language; there are no "optional extras."

Why was this done? The Department of Defense decided that the existence of language dialects would seriously hamper portability. In other words, if different dialects of Ada provide different subsets of its facilities, then it would be possible to port a program only if the destination Ada compiler provided the facilities used in the program.

Another reason for disallowing subsets was the discipline it imposed on the language designers. Language designers would be less likely to include highly complex, expensive, or hard-to-implement features in a language if they knew that such features would have to be provided in every implementation. It also discouraged them from designing in features that they were not sure how to implement (as happened in Algol-68).

We have seen that Ada 95 has relaxed these restrictions. The "core language" still must be implemented in its entirety, but there are six optional "annexes," which define, in effect, standard supersets of the core language.

Ada Has Been Criticized for Its Size

Ada is not a small language. Although there is no accepted way to measure the size of languages, it is significant that while Ada 83's context-free grammar is some 1600 tokens long, Pascal's is about 500 and Algol-60's is about 600. Of course, Ada provides some important facilities not included in Pascal and Algol-60, such as packages and tasks. But this does not necessarily mean that Ada's size is acceptable.

Ada 95 is even larger, due to its inclusion of object-oriented programming facilities and many other extensions. In addition, grammar-based measures such as these ignore the size of Ada's standard library, which is huge compared to other languages'.

Some computer scientists, such as C. A. R. Hoare, have suggested that Ada is so big that few programmers will ever master it. In his 1981 Turing Award paper, he said, "Do not allow this language in its present state to be used in applications where reliability is critical. . . ." Hoare has suggested that "by careful pruning of the Ada language, it is still possible to select a very powerful subset that would be reliable and efficient in implementation and safe and economic in use." He makes an important observation: "If you want a language with no subsets, you must make it *small*." The size of Ada remains an issue of vigorous debate. These debates are reminiscent of the debates in the late 1960s and early 1970s about PL/I's size, which were partly responsible for the success of Pascal.

Ada Has Been Moderately Successful

Since its introduction in 1983 and the development of reliable compilers in the late 1980s, Ada 83 has become widely used by the U.S. Department of Defense and related industries. However, Ada has not been as successful in universities and the consumer software industry, where less secure languages such as C and C++ have become popular. Although comparison studies have shown that Ada programs are significantly less expensive to develop and maintain than C or C++ programs, use of the latter languages has been encouraged by social factors other than the economics of program development. Among these factors are the widespread use of Unix (programmed in C) in universities (especially prestigious ones) and the comparatively small economic costs to the manufacturer of consumer software failures. We can expect Ada to remain an important language in defense-related industries, but it is not clear how well it will establish itself outside that application area.

Featuritis Is a Trap for the Language Designer

There is a kind of entropy in the evolution of programming languages that causes even the best designs to degenerate over time. This problem is sometimes called "featuritis" or the "Creaping Feature Syndrome"; it is especially prevalent in languages that are designed or maintained by committees. Featuritis arises from the fact that it is inherently easier to defend adding a feature to a language than to oppose it. This is because the benefits of adding a feature often outweigh the consequent increase in complexity, and because the benefits are specific and clear, whereas the costs are often general and hidden. (For an example, review our discussion of adding a **complex** type to Algol-60, Section 3.4.) However, the cumulative effect of many such decisions, made independently and without regard for each other, can be a large, ungainly language with rampant feature interaction. Ironically, it often happens that the excessive complexity of the language is eventually recognized, which motivates the designers to reject genuinely worthwhile features, which could have been included had less useful features been rejected.

Resisting the Creaping Feature Syndrome demands enormous discipline (and stubbornness!) from the individual or committee responsible for the final design of the language. They must be willing to turn their backs on individually attractive features for the sake of the complete language. One way of enforcing this discipline is to adopt at the start some absolute maximum complexity for the language, as measured by grammar size, for example. Then, once this complexity limit is reached, no other features can be added unless already adopted

features are eliminated or simplified. Such methods are widely used in other engineering disciplines; for example, absolute weight and power consumption limits may be placed on embedded computers.[5]

Featuritis is also combatted by having separate groups responsible for feature design and language design; with the entire language in their view, the language designers can select the most useful features from an array of available features, and integrate them into a unified whole. Several language designers have noted that somewhat orthogonal skills are required for feature design and language design. Pascal illustrates how the two complement each other: Hoare designed many of the individual features (e.g., finite sets, records, labeled **case**-statements); Wirth combined them into an integrated language.

Featuritis has even greater virulence in the *evolution* of programming languages, for once a feature has been included in a language and programmers have used it, it is nearly impossible to eliminate. Indeed, there are important economic reasons for "upward compatibility." Thus we have the extremely slow process of eliminating features from a standardized language such as FORTRAN (discouraging their use in one standard before they can be eliminated in the next). FORTRAN began as a simple language, but has become more complex in time. Likewise, Algol. Therefore Pascal was designed as a simple language, but it has also grown in complexity. Ada began as a complex language, but nevertheless has become even more complex (see especially Section 12.6 on object-oriented extensions to Ada). As a consequence, often the only solution to featuritis is to start over from scratch (e.g., Wirth's design of Pascal).

Characteristics of Fourth-Generation Programming Languages

Some of the characteristics of the fourth generation[6] are simply a consolidation and correction of certain of those of the third generation. In other respects the fourth-generation languages provide important new facilities, such as linguistic support for information hiding and concurrent programming. We consider in turn each of the structural domains.

The most important contribution of the fourth generation lies in the domain of *name structures*. In fact, *fourth-generation programming language* is essentially synonymous with *data abstraction language*, since the primary characteristic of this generation is the provision of an encapsulation facility supporting the separation of specification and definition, information hiding, and name access by mutual consent. Most of these languages allow encapsulated modules to be generic (or polymorphic), thus leading to an operator identification problem such as we saw in Ada. The optimum mix between flexibility and simplicity has yet to be determined (see also Chapter 12).

The second area in which the fourth generation is distinctive is *control structure*, since it is characteristic of this generation to provide for concurrent programming. Most fourth-generation languages use some form of message passing as a means of synchronization and

---

[5] For more on programming language metrics, see B. J. MacLennan, "Simple Metrics for Programming Languages," *Inform. Process. Manage. 20*, 1–2 (1984), pp. 209–221.

[6] We are discussing here the characteristics of fourth-generation *programming* languages. The term "fourth-generation language" is sometimes used to refer to application generator programs, which might or might not be programming languages in the technical sense discussed in the first two pages of the Introduction.

communication among concurrent tasks. Protected data structures, such as Ada 95's protected types, are also typical. On the other hand, the basic framework of these languages is still sequential. Fourth-generation languages typically also have a dynamically scoped exception mechanism for handling both system- and user-defined errors.

The data structure *constructors* of this generation are similar to those of the third generation, except that some problems (e.g., array parameters) have been corrected. Name equivalence is the rule in the fourth generation, although there are numerous exceptions to make it secure yet convenient. The primitive data structures tend to be more complicated than in the third generation, because of the desire to control accuracy and precision in numeric types.

Finally, the *syntactic structures* of the fourth generation are largely those of the second and third, that is, they are in the Algol/Pascal tradition. The major exception is a preference for fully bracketed structures.

In summary, the fourth generation can be seen as the culmination and fulfillment of the evolutionary process that began with FORTRAN. Although these languages are far from perfect, it is difficult to see how substantial improvements can be made to them without a radical change of direction. Does this mean that programming language evolution is at an end? Hardly.

Fifth-Generation Programming Languages

We have now entered the *fifth generation* of programming language design. Nobody yet knows what the dominant programming ideas of this generation will be. Many bold experiments are in progress, but their outcome is still uncertain. Therefore, in the remainder of this book we investigate three possible candidates for the programming paradigm of the future. These are *function-oriented programming*, *object-oriented programming*, and *logic-oriented programming*. The use of the word "oriented" indicates that each of these paradigms is organized around a comprehensive view of the programming process. Each attempts to push its view to the limit. Which, if any, will become dominant it is too early to say. Perhaps it will be some synthesis of them all.

EXERCISES*

1. Attack or defend this statement: Position-independent arguments are worthless; no procedure should have so many parameters that it needs them, since a large number of parameters results in a very wide interface.

2. Ada is unique among programming languages in that the designers have recorded many of the reasons for their design decisions ("Rationale for the Design of the ADA Programming Language," *SIGPLAN Notices 14*, 6, June 1979; Intermetrics, *Ada 95 Rationale*, 1995). Write a critique of one chapter of one of the Ada Rationales.

3. Compare the tasking facilities of Ada with those of Concurrent Pascal (P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Trans. Software Engineering 1*, 2, June 1975).

4. In the Ada 83 Rationale it is asserted that *path expressions* (R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science 16*, Springer, 1974) can be easily expressed in Ada. Show that this is true.

5. Read and critique Hoare's "Communicating Sequential Processes" (*Commun. ACM 21*, 8, August 1978).

6. Read and critique J. B. Goodenough's "Exception Handling: Issues and a Proposed Notation" (*Commun. ACM 18*, 12, December 1975). Compare the proposal in this paper with Ada's mechanism.

7. Ada's use of a semicolon as a terminator rather than a separator is based on research by Gannon and Horning ("Language Design for Programming Reliability," *IEEE Trans. Software Engineering 1*, 2, June 1975). Critique their paper.

8.** Read and critique the revised "IRONMAN" specifications ("Department of Defense Requirements for High Order Computer Programming Languages," *SIGPLAN Notices 12*, 12, December 1977).

9.** Read the revised "IRONMAN" specifications (see Exercise 8) and decide how well Ada 83 met them.

10. Read and critique Hoare's Turing Award Paper ("The Emperor's Old Clothes," *Commun. ACM 24*, 2, February 1981).

11.** Read and critique a proposal for decreasing Ada's size (e.g., H. F. Ledgard and A. Singer, "Scaling Down Ada [Or Towards a Standard Ada]," *Commun. ACM 25*, 2, February 1982).

12.** Shaw, Hilfinger, and Wulf designed the language Tartan to "determine whether a 'simple' language could meet substantially all of the Ironman requirement." Read and critique the papers on Tartan ("TARTAN—Language Design for the Ironman Requirement," *SIGPLAN Notices 13*, 9, September 1978).