2 EMPHASIS ON EFFICIENCY: FORTRAN

2.1 HISTORY AND MOTIVATION

Automatic Coding Was Considered Unfeasible

In the early 1950s there was little sympathy for the idea of a high-level programming language.[1] Indeed, many programmers were even opposed to the use of decimal numbers in programming! An elegant algebraic language developed by Laning and Zierler of MIT was compiling code as early as 1952, but it was largely ignored. Simple assemblers and libraries of subroutines were the accepted tools of the day. Simple interpreters continued to be used since the overhead of simulating floating-point arithmetic masked the overhead of interpretation. John Backus of IBM had designed one such system, called Speedcoding, which was very popular among IBM 701 installations. Backus's recognition that it already was more expensive to design and debug a program than to run it led to his development of Speedcoding and to his suggestions to include floating-point arithmetic and indexing in the IBM 704. When the 704 with its floating-point hardware was released, it exposed the inherent overhead of the interpretive routines and made them much less attractive. These factors led Backus to conclude that programming costs could be decreased only by a system that allowed the programmer to write in conventional mathematical notation and that generated code whose efficiency was comparable to that produced by a good programmer. In late 1953 Backus proposed these ideas to his management at IBM, and in January 1954, with one assistant, he began work on what was to become FORTRAN.

Preliminary FORTRAN Was Designed

In the 1950s Grace Murray Hopper, another pioneer language developer, organized a number of symposia under the auspices of the Office of Naval Research (ONR). At the ONR

[1] Historical information in this chapter comes mostly from the "History of Programming Languages Conference Proceedings," *SIGPLAN Notices 13*, 8 (August 1978). See Backus (1978).

symposium of May 1954, Backus and a colleague presented a paper on Speedcoding in which they discussed some of their ideas for a programming language based on mathematical notation. They were given a copy of a report describing the Laning and Zierler system, which was demonstrated for them in June 1954. By November 1954 Backus and three associates had produced a preliminary external specification for "the IBM Mathematical FORmula TRANslating system, FORTRAN." At a 1978 conference on the history of programming languages, Backus stated, "As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely as a simple preclude to the real problem: designing a compiler which could produce efficient programs." According to Backus, the preliminary specification of FORTRAN was met with "indifference and skepticism." Dystopian attitudes prevailed.

The FORTRAN Compiler Was Successful

Backus and his team, which eventually included nine others, began the design and programming of FORTRAN in early 1955 and released the system in April 1957 after 18 man-years of work. By most accounts the system did not really work at this time, although by April 1958 it had been enthusiastically accepted. At the 60 IBM 704 installations in existence in late 1958, more than half of the instructions were being generated by FORTRAN. In part this can be attributed to the exceptionally clear documentation that accompanied the FORTRAN system. It was also because of some very sophisticated optimization techniques, which Backus claims were not surpassed until the late 1960s. These optimizations allowed the FORTRAN system to deliver the efficiency that had been promised.

FORTRAN Has Been Revised Several Times

The experience gained with this FORTRAN system led Backus and his colleagues to propose FORTRAN II in September 1957. The compiler was available in the spring of 1958, and the language remains in use to this day. A dialect called FORTRAN III was designed and implemented in late 1958, but it never achieved widespread use because of its many dependencies on the IBM 704. In 1962 the FORTRAN IV language was designed; it is still an important FORTRAN dialect. In *Programming Languages: History and Fundamentals*, Jean Sammet said, "By 1963 virtually all manufacturers had either delivered or committed themselves to producing some version of FORTRAN." Reflecting FORTRAN's popularity, the American National Standards Institute (ANSI) began development of standard FORTRAN IV (ANS FORTRAN) in 1962; this was completed in 1966. Even after this, dialects of FORTRAN were very common and few compilers implemented exactly the ANSI standard. In 1977 a new ANS FORTRAN was developed that is sometimes known as FORTRAN 77. This language incorporates a number of ideas from later languages, although it is still a 1950s language in its capabilities. After a 12-year development process, a new standard, commonly known as FORTRAN 90, was produced. Although it incorporates additional modern features from newer languages, it is a relatively minor revision. FORTRAN 2000 is already on the drawing board.

In any case, we will concentrate on the 1966 ANS FORTRAN since our intent is to illustrate the characteristics of first-generation languages, and these are more apparent in FOR-

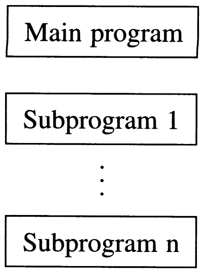
TRAN IV. Unless otherwise specified, in this chapter the name FORTRAN refers to ANS FORTRAN IV.

2.2 DESIGN: STRUCTURAL ORGANIZATION

Figure 2.1 displays a small FORTRAN I program to compute the average of the absolute value of an array. (Since we are trying to illustrate a number of FORTRAN's features, we have not written the best program possible to do this job.) Although we have used FORTRAN I for the example so that the similarity to pseudo-codes is more apparent, this example is still legal FORTRAN 77.

Programs Are Divided into Disjoint Subprograms

The pseudo-code we discussed in Chapter 1 and the preliminary FORTRAN specification both lacked what is now considered to be a crucial part of programming languages: subprograms (e.g., procedures, functions, and subroutines). By the time the FORTRAN I system was released, a subprogram facility had been included. The overall structure of a FORTRAN program is a *main program* and zero or more *subprograms*, for example,



Subprograms will be discussed in detail in Section 2.3; suffice it to say that they can communicate using *parameters* or through shared data areas called *COMMON blocks.*

Constructs Are Either Declarative or Imperative

We saw in Chapter 1 that pseudo-code programs were divided into two parts: first, a *declarative* part, which described the data areas, their lengths, and their initial values; second, an *imperative* part, which contained the commands to be executed during the running of the program. Declarative and imperative constructs occur in almost all programming languages, although they may be called by different names. For example, in FORTRAN the declarative constructs are often called *nonexecutable statements* and the imperative constructs are often called *executable statements.*

Declarative constructs are like declarative statements in natural languages: They state facts about the program, which are used at *compile-time.* Imperative constructs are like im-

```
      DIMENSION DTA(900)
      SUM = 0.0
      READ 10, N
10    FORMAT(I3)
      DO 20 I = 1, N
      READ 30, DTA(I)
30    FORMAT(F10.6)
      IF (DTA(I)) 25, 20, 20
25    DTA(I) = -DTA(I)
20    CONTINUE
      DO 40 I = 1, N
      SUM = SUM + DTA(I)
40    CONTINUE
      AVG = SUM/FLOAT(N)
      PRINT 50, AVG
50    FORMAT(1H , F10.6)
      STOP
```

Figure 2.1 A FORTRAN I Program (also Legal FORTRAN 77)

perative statements in natural languages: They give a command, which the program obeys at *run-time*. Next, we briefly survey the FORTRAN constructs of each of these types.

Declarations Include Bindings and Initializations

In our symbolic pseudo-code we saw that the declarations performed three functions:

1. They *allocated* an area of memory of a specified size.
2. They attached a symbolic name to that area of memory. This is called *binding* a name to an area of memory.
3. They *initialized* the contents of that memory area.

These are three important functions of declarations in most languages, including FORTRAN. For example, the declaration

```
DIMENSION DTA (900)
```

causes the loader to allocate 900 words and to bind the name DTA to this area. A separate kind of declaration, called a DATA declaration, can be used for initialization. For example,

```
DATA DTA, SUM / 900*0.0, 0.0
```

would initialize the array DTA to 900 zeroes and the variable SUM to zero. FORTRAN does not require the programmer to initialize storage; this lack of initialization is a frequent cause of errors. Declarations and initializations are discussed in Section 2.3.

Imperatives Are Either Computational, Control-Flow, or Input-Output

In our pseudo-code we saw that there were three different kinds of imperative statements:

1. Computational statements, such as the arithmetic and the move operations.
2. Control-flow statements, such as the comparisons and the looping statements.
3. Input-output statements, namely, `READ` and `PRINT`.

FORTRAN provides imperatives of exactly the same three types. The primary computational statement is the *assignment* statement, for example,

```
AVG = SUM / FLOAT(N)
```

The assignment statement is also one of the contexts in which *algebraic expressions* can appear. Recall that the ability to use conventional algebraic notation was one of the important contributions of FORTRAN.

Corresponding to the comparisons in the pseudo-code, there are *conditionals*, which are called `IF`-statements in FORTRAN. Similarly, corresponding to the looping instruction of the pseudo-code, FORTRAN has a `DO`-loop. Finally, FORTRAN provides an unconditional transfer instruction, the `GOTO`-statement, which we did not put in the pseudo-code (because it was easily expressed by the equality test). We can see that FORTRAN does not go too far beyond the pseudo-code in its control-flow statements.

FORTRAN has a much more elaborate set of input-output instructions than did our pseudo-code. In fact, even FORTRAN I had 12 statements for performing or controlling input-output out of a total of only 26 statements. Some of these were for handling a wider variety of devices, such as tapes and drums, and some of these were for providing more explicit control over the format of data on the input-output media. FORTRAN's imperative statements will be discussed in Section 2.3.

A Program Goes Through Several Stages to Be Run

As we have seen, one of the most important goals for FORTRAN was efficient execution; therefore, the overhead for interpretation was completely unacceptable. The approach adopted, and still used in most FORTRAN systems, is that the program progresses through a series of stages:

1. Compilation
2. Linking
3. Loading
4. Execution

The first step, *compilation*, translates individual FORTRAN subprograms into *relocatable object code.* That means that by a process similar to the pseudo-code translator, the FORTRAN statements are translated into the instructions, or *object code*, of a real computer. Since each subprogram must reside in memory with other subprograms that may not have yet been compiled, it is impossible to determine at compile-time the exact location in memory in which each subprogram will go. Therefore, the exact addresses of variables and statements are not yet known; we say that they have a later *binding time* because they are bound to addresses at load-time rather than at compile-time. It is for these reasons that the object program is represented in a special *relocatable* format that allows the addresses to be assigned at load-time. Relocatable format is similar to the symbolic statement and variable labels in our pseudo-code.

The second step, *linking*, addresses the need for incorporating *libraries* of already programmed, debugged, and compiled subprograms. Needless to say, the presence of a good library can greatly simplify the programming process, as we saw with floating-point libraries in Chapter 1. The use of libraries means that programs contain *external references* to subprograms in these libraries. Furthermore, these library subprograms may themselves contain external references to other library subprograms. To obtain a complete program, all of these external references must be resolved or *satisfied* by finding the corresponding subprograms in the library. This is the goal of the linking process; its result is usually a file containing all of the parts of the program, still in relocatable format, but with their external references satisfied.

The third step, *loading*, is the process in which the program is placed in computer memory. This requires converting it from relocatable to *absolute* format, that is, it requires binding all code and data references to the addresses of the locations that the code and data will occupy in memory.

The final step, *execution*, is the one in which control of the computer is turned over to the program in memory. Since the program is executed directly rather than interpreted, it has the potential of running much faster.

Compilation Involves Three Phases

The compilation process is obviously one of the most important steps in the processing of a FORTRAN program since it is this step that determines the efficiency of the final program. For a language such as FORTRAN, compilation usually involves three tasks. These may be performed one after the other or interleaved in various ways.

1. **Syntactic analysis:** The compiler must classify the statements and constructs of FORTRAN and extract their parts.
2. **Optimization:** As we have said, efficiency was a prime goal of the original FORTRAN system. For this reason the original FORTRAN system included a sophisticated optimizer whose goal was to produce code as efficient as could be produced by an experienced programmer. Most FORTRAN compilers perform at least a moderate amount of optimization.
3. **Code synthesis:** The final task of compilation is to put together the parts of the object code instructions in relocatable format.

2.3 DESIGN: CONTROL STRUCTURES

Control Structures Govern Primitive Statements

In Section 2.3 we discuss FORTRAN's *control structures*, that is, those constructs in the language that govern the flow of control of the program. We will find that these control structures are elaborations of the control structures found in the pseudo-code of Chapter 1. In the pseudo-code we saw that the purpose of control structures was to direct control to various *primitive* computational and input-output instructions such as ADD and READ. (By a *primitive* operation we mean one that is not expressed in terms of more fundamental ideas in the language.) The situation is similar in FORTRAN; the computational and input-output instructions do the actual data processing work of the program, while the control structures act as "schedulers" or "traffic managers" by directing control to these primitive statements. We will see in later chapters that this organization is common to all *imperative* programming languages.

As we have said (Section 2.1), the ability to write more or less familiar looking algebraic equations was one of the major contributions of FORTRAN. Without doubt, the assignment statement is the most important statement in FORTRAN. In fact, a FORTRAN program can be considered as a collection of assignment statements with provision (i.e., the control structures) for directing control to one or the other of these assignment statements.

Control Structures Were Based on IBM 704 Branch Instructions

FORTRAN was originally designed as a programming language for the IBM 704 computer; it was thought that there would be similar, but different, languages for other computers. Backus has said that he never imagined that FORTRAN would be used on the computers of other manufacturers. It is thus not surprising that the first FORTRAN had many similarities to the 704 instruction set; designers have a tendency to include in a language the features they have previously found useful. This *machine dependence* is a characteristic of first-generation languages.

We can see the machine dependence of FORTRAN's control structures in Figure 2.2, which displays the similarities between FORTRAN II's control structures and the branch instructions of the IBM 704. It is not important that you understand the 704 instructions or even the FORTRAN statements at this point; what is important is the correspondence. It is one reason that FORTRAN has sometimes been called an "assembly language for the IBM

FORTRAN II Statement	704 Branch	
`GOTO n`	`TRA k`	(transfer direct)
`GOTO n, (n1, n2, ..., nm)`	`TRA i`	(transfer indirect)
`GOTO (n1, n2, ..., nm), n`	`TRA i,k`	(transfer indexed)
`IF (a) n1, n2, n3`	`CAS k`	(compare AC with storage)
`IF ACCUMULATOR OVERFLOW n1, n2`	`TOV k`	(transfer on AC overflow)
`IF QUOTIENT OVERFLOW n1, n2`	`TQO k`	(transfer on MQ overflow)
`DO n i = m1, m2, m3`	`TIX d,i,k`	(transfer on index)
`CALL name (args)`	`TSX i,k`	(transfer and set index)
`RETURN`	`TRA i`	(transfer indirect)

Figure 2.2 Similarity of FORTRAN and IBM 704 Branches

704," although some of the more blatantly machine-dependent statements (e.g., `IF QUOTIENT OVERFLOW`) were removed from FORTRAN IV and later versions. This correspondence also explains some of FORTRAN's more unusual control structures, for example, the *arithmetic* `IF`*-statement*

IF (e) n_1, n_2, n_3

evaluates the expression e and then branches to n_1, n_2, or n_3 depending on whether the result of the evaluation is negative, zero, or positive, respectively. This is exactly the function of the 704's `CAS` instruction, which compares the accumulator with a value in storage and then branches to one of three locations. The arithmetic `IF` was not very satisfactory for a number of reasons, including the difficulty of keeping the meaning of the three labels straight and the fact that two of the labels were usually identical (because two-way branches are more common than three-way branches). In later versions of FORTRAN, a more conventional *logical* `IF`*-statement* was added, for example,

```
IF (X .EQ. A(I)) K = I - 1
```

(FORTRAN uses `.EQ.` for the equality relation.)

Machine-dependent features such as the arithmetic `IF` are violations of

The Portability Principle

Avoid features or facilities that are dependent on a particular computer or a small class of computers.

The GOTO Is the Workhorse of Control Flow

Just as in most computers, the transfer (i.e., branch or jump) instruction is the primary means for controlling the flow of execution, so in FORTRAN the GOTO-statement and its variants are the fundamental control structures. We will now investigate the implications of this fact on program readability.

In FORTRAN, the GOTO-statement is the raw material from which control structures are built. For example, a two-way branch is often implemented with a logical `IF`-statement and a GOTO:

```
          IF (condition) GOTO 100
          ... case for condition false ...
          GOTO 200
100       ... case for condition true ...
200       ...
```

We can see that this corresponds to the **if-then-else,** or *conditional* statement of newer programming languages, although the false- and true-branches are placed in the opposite order. If we wish to put them in the same order, we must then negate the condition, which may be confusing:

```
        IF (.NOT. (condition) ) GOTO 100
        ... case for condition true ...
        GOTO 200
100     ... case for condition false ...
200     ...
```

This use of the IF-statement divides the control flow into two cases. For dividing it into more than two cases, a *computed* GOTO is provided, for example,

```
        GOTO (10, 20, 30, 40), I
10      ... handle case 1 ...
        GOTO 100
20      ... handle case 2 ...
        GOTO 100
30      ... handle case 3 ...
        GOTO 100
40      ... handle case 4 ...
100     ...
```

The meaning of this is to branch to statement 10, 20, 30, or 40 if I is 1, 2, 3, or 4, respectively. We can recognize this as the equivalent of the **case**-statement that is included in many contemporary languages.[2]

Both the IF-statement and the computed GOTO are examples of *selection* statements, so called because they select between two or more possible control paths.

Loops can be implemented by various combinations of IF-statements and GOTOs (the DO-loop is discussed later). For example, a *trailing-decision loop* could be written

```
100     ... body of loop ...
        IF (loop not done) GOTO 100
```

and a *leading-decision loop* could be written

```
100     IF (loop done) GOTO 200
        ... body of loop ...
        GOTO 100
200     ...
```

We can recognize these as the **while-do** and **repeat-until** constructs of languages such as Pascal. These constructs are called *indefinite iterations* because the exact number of iterations is not known in advance (i.e., not definite).

One problem with using one statement (GOTO) to build all control structures is that we never know what control structure is intended. For example, in Pascal, when we see **while** we know that a leading-decision loop is intended; when we see **repeat** a trailing-decision loop is intended; and when we see **if** a conditional statement is intended. In FORTRAN, when we see an IF-statement, we don't know (without looking closely) whether it's the beginning of a leading-decision loop, the end of a trailing-decision loop, a conditional selec-

[2] The labels is a computed GOTO need not be unique.

tion, or part of a more complicated control structure (such as the mid-decision loop, discussed below). This difficulty in identifying structures makes it much harder for a reader to determine the programmer's intent.

▨ ***Exercise 2-1:*** Use arithmetic `IF`s and assignment statements to accomplish the following: store +1 in S if $X > 0$, store -1 in S if $X < 0$, and store 0 in S if $X = 0$. Do the same with logical `IF`s. (Note that in FORTRAN '<' and '>' are written '`.LT.`' and '`.GT.`', respectively.)

▨ ***Exercise 2-2:*** Write in FORTRAN IV a leading-decision loop that doubles N until it is greater than 100.

▨ ***Exercise 2-3:*** Write FORTRAN IV code to compute the first Fibonacci number greater than 1000.

▨ ***Exercise 2-4:*** Translate '`GOTO (10, 20, 30, 40),I`' into `IF`-statements.

▨ ***Exercise 2-5:*** Write a computed `GOTO` that, on the basis of a number `M` representing a month, branches to label `100` if the month is February, to label `200` if the month has 30 days, and to label `300` if the month has 31 days.

It is Difficult to Correlate Static and Dynamic Structures

Of course, the patterns shown above are not the only ways of combining `IF` and `GOTO` statements in FORTRAN. It is possible to write *mid-decision loops*,

```
100     ... first half of loop ...
        IF (loop done) GOTO 200
        ... second half of loop  . . .
        GOTO 100
200     . . .
```

and much more complicated control structures. The `GOTO`-statement is a very primitive and powerful control structure. It is a two-edged sword because it is possible to implement almost any control regime with it—those that are good, but also those that are bad.

What makes a control regime good or bad? Mainly it is understandability; in a good control structure the static form of the structure (i.e., its appearance to the reader) corresponds in a simple way to its dynamic behavior. Therefore, it is easy for a reader to visualize the effect of a control structure from its written form. The undisciplined use of the `GOTO`-statement permits the construction of very intricate control structures, which are correspondingly hard to understand. We will see in Chapters 3–5 that newer languages (including FORTRAN 77) depend much less on the `GOTO`-statement.

The idea of a good control structure is embodied in the Structure Principle first proposed by E. W. Dijkstra[3]:

[3] See Dijkstra (1968).

The Structure Principle

The static structure of a program should correspond in a simple way to the dynamic structure of the corresponding computations.

This principle means that it should be possible to visualize the behavior of the program easily from its written form. For example, when the execution of one segment of code precedes another in time, the statements of the first segment should precede those of the second in the program. Similarly, the statements whose execution is repeated by a loop should be easy to identify. This is simplified if they are a contiguous, indented block of text. We will see many other examples of the Structure Principle in this book.

▩ ***Exercise 2-6:*** Suggest a practical use for mid-decision loops.

The Computed and Assigned GOTOs Are Easily Confused

Two statements in FORTRAN, the *computed* GOTO and the *assigned* GOTO, illustrate the pitfalls that await the language designer. We have seen the computed GOTO:

GOTO ($L_1, L_2, ..., L_n$), I

where the L_i are statement numbers and I is any integer variable. The computed GOTO transfers to statement number L_k if I contains k. Computed GOTOs are usually implemented by a jump table: The compiler stores addresses (of the statements numbered L_i) in an array in memory and then compiles code to use I as an index into this array.

FORTRAN also provides another control structure for branching to a number of different statements, the *assigned* GOTO:

GOTO N, ($L_1, L_2, ..., L_n$)

where N is also an integer variable. This statement transfers to the statement whose *address* is in the variable N; in other words, this is an indirect GOTO. The list of statement labels is not actually necessary since all the information necessary to perform the jump is in N. The list is provided as documentation since otherwise the reader would have no way of knowing where the GOTO goes. Most compilers do not check whether the statement whose address is in N has its label included in the list (since this check would be comparatively expensive).

The assigned GOTO must be used in conjunction with another statement, the ASSIGN statement. The effect of

```
ASSIGN 20 TO N
```

is to put the address of statement number 20 in the integer variable N. Note that this is completely different from the assignment statement N = 20, which stores the integer 20 into N. In general, the address of statement number 20 will not be 20 (recall that the symbolic labels in our pseudo-code had no relation to the addresses to which they were bound). Thus, the effect of `ASSIGN 20 TO N` will be to put some other number (the address of statement 20, say 347) into `N`.

The computed and assigned GOTOs are easily confused; they look almost identical. Therefore, it is not uncommon for a programmer to write one where the other is expected. Let us consider the consequences of writing a computed GOTO where an assigned GOTO is intended:

```
ASSIGN 20 TO N
   .
   .
   .
GOTO (20, 30, 40, 50), N
```

The `ASSIGN`-statement will assign the address of statement number 20 (say, 347) to `N`. The computed `GOTO` will then attempt to use this as an index into the jump table (20, 30, 40, 50). In this case, the index (347) will be well out of range, but most systems do not check this so we will fetch some value out of memory to use as the destination of the jump. The result is that the program will transfer to an unpredictable location in memory, thus leading to a very-difficult-to-find bug.

Now let's consider the opposite error: using an assigned GOTO where a computed GOTO is intended.

```
I = 3
   .
   .
   .
GOTO I, (20, 30, 40, 50)
```

Since the assigned GOTO expects the variable `I` to contain the address of a statement, it will transfer to that address. In this case, it will transfer to address 3, which is almost certainly not the address of one of the statements in the list and is very likely not the address of a statement at all (low-addressed locations are often dedicated to use by the system). Again, a difficult bug results.

What are the causes of these problems? The most obvious cause is the easily confused syntax of the two constructs:

GOTO $(L_1 \ldots, L_n)$, `I`

GOTO `I`, $(L_1, \ldots, L_n)$

This is a violation of the Syntactic Consistency Principle.

Syntactic Consistency Principle

Things that look similar should be similar and things that look different should be different.

Generally speaking, it is best to avoid syntactic forms that can be converted into other legal forms by a simple error. (FORTRAN's use of '`**`' for exponentiation has been criticized on this basis, since leaving out one of the asterisks converts it into a legal FORTRAN multiplication sign '`*`'.)

A more fundamental cause for the GOTO problem is FORTRAN's *weak typing*, which results from using integer variables to hold a number of things besides integers, such as the addresses of statements, and character strings (discussed in Section 2.4, pp. 69–70). If FORTRAN had variables of type LABEL for holding the addresses of statements, and if a LABEL variable were required in an assigned GOTO, then confusing the two GOTO statements would lead to an easy-to-find compile-time error not to an obscure run-time error. Thus, FORTRAN violates the principle of Defense in Depth.

Defense in Depth Principle

If an error gets through one line of defense (syntactic checking, in this case), then it should be caught by the next line of defense (type checking, in this case).

This example also illustrates one of the hardest problems of language design: identifying the *interaction* of features. (In this case, the interaction is between the syntax of the GOTOs and the decision to use integer variables to hold the addresses of statements.)

- ***Exercise 2-7*:*** Explain why it is computationally expensive to check whether the destination of an assigned GOTO is in its list of statement labels.

- ***Exercise 2-8*:*** Propose and defend a solution to the problem of assigned and computed GOTOs in FORTRAN. If you decide to introduce a type LABEL, keep in mind that you must analyze the interaction of this feature with others already in the language. For example, will you allow LABEL arrays? Will you allow parameters of type LABEL or allow functions to return LABELs? Discuss the pros and cons.

The DO-Loop Is More Structured than the GOTO

We have seen that the GOTO- and IF-statements provide *primitive*, or *low-level*, control structuring mechanisms. That is, they are simple components from which higher-level control structures can be built. FORTRAN contains only one built-in, higher-level control structure, the DO-loop, which we discuss next. Much like the LOOP instruction in our pseudo-code, the DO-loop provides a simple method of constructing a counted loop (sometimes called a *definite iteration*). For example,

```
      DO 100 I = 1, N
      A(I) = A(I)*2
100   CONTINUE
```

is a command to execute the statements between the DO and the corresponding CONTINUE with I taking on the values 1 through N. The variable that changes values (I in this case) is called the *controlled variable*, and the statements that are repeated, which extend from the DO to the CONTINUE with a matching label, are called the *extent* or *body* of the loop.

- ***Exercise 2-9:*** Write the above example using only IF and GOTO (i.e., without the DO-loop).

- ***Exercise 2-10:*** Write a DO-loop that computes into SUM the sum of the array elements A(1) + A(2) + ··· + A(100).

The DO-Loop Is Higher Level

The DO-loop is called *higher level* because it allows programmers to state what they *want* (namely, for the commands in the loop to be executed N times), rather than how to do it (e.g., initialize I, increment it, test it). Higher-level constructs such as this are safer to use because they remove from the programmer the opportunity to make certain errors, such as incorrect testing for loop termination. Thus, the DO-loop illustrates

The Impossible Error Principle

Making errors impossible to commit is preferable to detecting them after their commission.

It also illustrates the Automation Principle because we have made the computer take over a routine, tedious, error-prone task.

The DO-Loop Can Be Nested

One of the important characteristics of the DO-loop is that it can be nested; that is, DO-loops are constructed *hierarchically*. More complicated loops can be built up from simpler ones by nesting DO-loops one within another, for example,

```
          DO 100 I = 1, M ─────────┐
          :                        │
            DO 200 J = 1, N ──┐    │
            :                 │    │
200         CONTINUE ─────────┘    │
          :                        │
100       CONTINUE ────────────────┘
```

Hierarchical structures such as these are very common in programming languages and in other contexts where complicated systems must be built or described. An important requirement of hierarchical structure is that the structures must be properly nested. For example,

```
          DO 100 I = 1, M ─┐
          :                │
          DO 200 J = 1, N ─│──┐
          :                │  │
100       CONTINUE ────────┘  │
          :                   │
200       CONTINUE ───────────┘
```

is incorrectly nested, as shown by the brackets to the right. Insisting on a hierarchical structure ensures that there is a *regular* flow of control whose dynamic structure corresponds to the static structure of the program (the Structure Principle). Unfortunately, this regularity is compromised by the fact that FORTRAN permits the programmer to jump into and out of DO-loops under certain circumstances.

Overall, FORTRAN has a linear rather than a hierarchical structure, which reflects the pseudo-codes and machine language from which it is derived. The statements are listed one after another and numbered in a way reminiscent of memory addresses rather than being nested like the statements in more modern languages. Other than the DO-loop, the only nesting permitted in FORTRAN statements is in the logical `IF`, although the statement controlled by the `IF` is only allowed to be a simple unconditional statement (e.g., an assignment or GOTO, not an `IF` or DO).[4]

The DO-Loop is Highly Optimized

In Section 2.1 we discussed the dystopian resistance that the first FORTRAN system faced; programmers were convinced that a compiler could not produce code as good as they could produce. It was therefore imperative that the FORTRAN system produce excellent code. The most troublesome area in this regard was the use of index registers for array subscripting. Programmers were able to see what variables would be used for indexing in a loop and thereby were able to keep their values in index registers; this allowed faster indexing than could be obtained by fetching their values from memory. Similarly, the index registers would be incremented and tested directly without being stored in memory. Producing code of comparable quality posed a significant challenge for the designers of the first FORTRAN compiler. However, by using sophisticated optimization techniques they were able to produce code better than most assembly language programmers. Some of these techniques are discussed in Section 2.4 (pp. 73–75).

The ability to optimize the DO-loop can be partly attributed to its *structure*, the fact that the *controlled variable* and its initial and final values are all stated explicitly along with the extent of the loop. All this useful information, which is provided explicitly by the DO-loop, must be discovered by the compiler if the more primitive `IF` and GOTO are used. Although this can be done, it is difficult to program the compiler to do it, and it is expensive at compile time. It is often the case that higher-level programming language constructs are easier to optimize than the lower-level ones. This is an example of the Preservation of Information Principle.

Preservation of Information Principle

The language should allow the representation of information that the user might know and that the compiler might need.

[4] FORTRAN 77 makes more consistent use of hierarchical structure, which is a characteristic of second-generation languages. We restrict our attention to FORTRAN IV because it is more characteristically first generation.

Subprograms Were an Important, Late Addition

It is surprising to realize that the preliminary description of FORTRAN I omitted what is now considered one of the most important programming language facilities—subprograms. Of course, FORTRAN programs made use of library procedures, such as those for input-output, and the mathematical functions (`SIN`, `COS`, etc.) just as our pseudo-code did. What FORTRAN I did *not* provide was the capability for programmers to define their own subprograms. This deficiency was remedied in FORTRAN II with the addition of the `SUBROUTINE` and `FUNCTION` declarations. In the rest of Section 2.3 we will discuss subprograms and their implementation.

Subprograms Define Procedural Abstractions

To see the need for subprograms, we will work through a very simple example. Suppose we have a program, which in outline form is

```
   .
   .
   .
DIST1 = X1 - Y1
IF (DIST1 .LT. 0) DIST1 = -DIST1
   .
   .
   .
DIFFER = POSX - POSY
IF (DIFFER .LT. 0) DIFFER = -DIFFER
   .
   .
   .
```

(In FORTRAN, `.LT.` means "less than.") We can see that essentially the same program fragment has been repeated twice, although with different variables. When we *abstract* out this fragment from the particular variables it mentions, we have

$d = x - y$

IF (d .LT. 0) $d = -d$

In the first occurrence, d, x, and y are `DIST1`, `X1`, and `Y1`; in the second they are `DIFFER`, `POSX`, and `POSY`.

Instead of repeating this sequence every time it is needed, we can define this *procedural abstraction* (or *control abstraction*) once, and then call for it to be used every time we need it. This definition is done with a subprogram declaration. In this case, the kind of subprogram we need is a *subroutine*, which is declared by:

SUBROUTINE *name* (*formals*)
... *body of subroutine* ...
RETURN
END

where *name* is the name to be bound to the subroutine and *formals* is a list of the names of the *formal parameters* (or dummy variables); these are the variables that will stand for different variables each time the subroutine is used. (Although the RETURN-statement normally occurs at the end of the subprogram, it is allowed anywhere a statement is allowed.)

A subroutine that embodies the procedural abstraction in our example is

```
SUBROUTINE DIST (D, X, Y)
D = X - Y
IF (D .LT. 0) D = -D
RETURN
END
```

This subroutine can then be *invoked* by a CALL-statement; for example, the program fragment with which we started can now be written

```
      .
      .
      .
CALL DIST (DIST1, X1, Y1)
      .
      .
      .
CALL DIST (DIFFER, POSX, POSY)
      .
      .
      .
```

When such a procedure is invoked, we say that the formals are *bound* to the actuals; that is, in the first case, we bind D to DIST1, X to X1, and Y to Y1. In the second we bind D to DIFFER, X to POSX, and Y to POSY. By this means variables in the *caller's* environment can be accessed by referring to the formal parameters in the *callee's* environment (the callee is the subprogram being called).

Executing the CALL-statement passes control from the caller to the callee. Executing a RETURN-statement in the callee passes control back to the caller, which resumes execution at the statement following the CALL.

The advantages of procedural abstraction are discussed next.

▩ ***Exercise 2-11:*** Define a subroutine PUTABS (X, Y) that computes the absolute value of X and stores it in Y.

Subprograms Allow Large Programs to Be Modularized

One obvious advantage of procedural abstraction is that it saves writing, as we saw in the example above. We can save ourselves from writing the same or a similar code sequence over and over again by defining it as a subprogram and calling it when needed. There is a much more important reason for defining procedural abstractions that derives from the nature of human perception. Humans seem to be able to keep only a few distinct things (about seven) in their minds at one time. One result of this is that the complexity of understanding a system's design (whether software or not) increases rapidly with the size of the system. One of the most common approaches used to solve large problems is to break them into smaller problems. This is basically the function of procedural abstraction. By defining a program as a set of relatively small subprograms, the total difficulty of the design process can be decreased since each subprogram can be written, debugged, and read as an independent unit. We have seen this idea before; it is called *modularizing* a program because the program (and the programming task) is divided up into a number of manageable *modules*. *Abstraction*, one of the most common methods of modularization, is the process in which we *abstract out* the common parts of a system. This is familiar from elementary algebra, where, for example, an expression can be simplified by factoring:

$$ax - bx = (a - b)x$$

These ideas are embodied in a very important principle:

The Abstraction Principle

Avoid requiring something to be stated more than once; factor out the recurring pattern.

▪ ***Exercise 2-12*:*** Give some additional examples, from outside of computer science, where modularization and the Abstraction Principle are valid.

Subprograms Encourage Libraries

There are other advantages to subprograms, such as *separate compilation*. Since subprograms are largely independent of one another, they can be separately translated to relocatable object code by the compiler and later combined by the linker. Hence, when one module of a program is modified, only that module has to be recompiled because it can later be linked with the already compiled versions of the other modules. In a large program this can mean a great saving of computer time.

A natural adjunct to this separate compilation is the idea of maintaining a *library* of already debugged and compiled useful subprograms. This allows a programmer to build upon the work of others. A well-designed library is an enormous aid in program development; indeed, many programs are written as no more than a series of calls on library procedures. This sort of modular, or "pre-fab," program construction is called *programming in the large* to contrast it with *programming in the small*, wherein many small components, the statements of a language, are assembled to make the program.[5]

▪ ***Exercise 2-13*:*** Investigate the contents of the libraries on your local computer system. Which application areas are best supported by libraries? Which are least supported?

Parameters Are Usually Passed by Reference

In our discussion of the `DIST` subroutine example on page 55, we described the action of `CALL` as though the actual parameters were substituted textually for the formal parameters. This is called the *copy rule* for subprogram invocation; that is, the body of the subprogram is copied into the place from which it was called, with the actual parameters substituted for the formals. While this is a handy way to think about the meaning of parameters, it is not

[5] These terms were introduced in DeRemer and Kron (1976).

the way they are in fact implemented. The copies would occupy too much space and it would take too much time to substitute the parameters. Next, we discuss the *parameter passing modes*, or ways in which parameters are actually passed, in FORTRAN.

FORTRAN permits subprogram parameters to be used for input or output or both. In the `DIST` example, the `X` and `Y` parameters were used as input values and the `D` parameter was used as an output value. If the values of `X` and `Y` had been altered by the subroutine, then these would have been both input and output parameters. How can these parameters be implemented? First, consider an output parameter: If a subprogram is to be able to write into a variable passed as an actual parameter (e.g., `DIFFER` in our example), then it must be passed the address of this variable's location in memory. In other words, the formal parameter `D` is bound to the *address* of the actual parameter `DIFFER`. Therefore, when the assignment `D = X - Y` is executed, it is `DIFFER` that is modified. Thus, the formal parameter gives the callee read-write access to the actual parameter in the caller. The technical term used in programming languages for an address of a location in memory is a *reference* because it *refers* to a memory location. For this reason the parameter passing mode we have just described is called *pass by reference*. Since FORTRAN does not permit programmers to specify whether they intend to use a parameter for input or output or both, all parameters are passed by reference just on the chance they may be output parameters. The consequences of this decision are discussed next.

Pass by Reference is Always Efficient

One of the advantages of pass by reference is that it is always efficient. To show this we investigate how several different kinds of actual parameters are passed. The key point is that pass by reference always passes just the reference to the actual parameter, that is, its address. Since references are usually small (one word or less), this means that very little information is passed from the *caller* to the *callee*. In the case of actuals that are simple variables, as in `CALL DIST (DIFFER, POSX, POSY)`, the addresses of the three actuals (`DIFFER`, `POSX`, and `POSY`) are passed.

Suppose that the actual parameters were array elements, for example,

```
CALL DIST (D(I), POS(I), POS(J))
```

The address of `POS(I)` is easily computed from the address of the first element of `POS` and the value of the index `I` (this process is described in the section on data structures, page 71). Therefore, all the caller has to do is compute the addresses of the array elements, `D(I)`, `POS(I)`, and `POS(J)`, and pass these to the caller, just as it did for simple variables.

We have seen how array elements are passed to subprograms; how about the entire array, which is often useful? For a simple example, consider the following `FUNCTION` that computes the mean (average) of an array. (A `FUNCTION` is a subprogram that computes a value and hence may be called from an expression.)

```
      FUNCTION AVG (ARR, N)
      DIMENSION ARR(N)
      SUM = 0.0
      DO 100 I = 1, N
         SUM = SUM + ARR(I)
```

```
100     CONTINUE
        AVG = SUM / FLOAT(N)
        RETURN
        END
```

A simple program that uses this function is

```
DIMENSION VALUES (100)
     :
AVGVAL = AVG (VALUES, 100)
     :
```

It is necessary to pass the size of the array to the subprogram because it is needed to control the DO-loop (among other uses). Unfortunately, having to pass the array size means that the programmer has the opportunity to pass the wrong size. This may introduce a bug into the program if the size passed is too small, or it may compromise the security of the FORTRAN run-time system if it is too big. This is so because FORTRAN permits programmers to oversubscript arrays, that is, to use array subscripts that are *out of bounds* (the reason this is permitted is discussed later). Thus, the programmer may inadvertently store into tables or other data structures that the compiler has placed in memory for the use of the run-time system. This causes the program either to abort or to behave in mysterious ways. Requiring the programmer to pass the array's size violates

The Impossible Error Principle

Making errors impossible to commit is preferable to detecting them after their commission.

Clearly, the array must be passed in such a manner that all of its elements are accessible to the callee. This is accomplished by passing the address of the first element of the array; that is, the formal ARR is bound to the first element of the array actual, VALUES(1). The callee can then compute the address of an array element ARR(I) in the usual way. This is a very efficient way of passing arrays since only a single address is passed from the caller to the callee. Unfortunately, this efficiency has bad consequences, some of which are discussed in the next section.

We must note that there is an efficiency *cost* associated with reference parameters: the cost of *indirection*. Instead of the subprogram having the value of the actual parameter directly available, it only has the address of the actual. Therefore, an extra memory reference is required to fetch or store the value of a parameter passed by reference. This is discussed further in Chapter 8 (Section 8.1).

Pass by Reference Has Dangerous Consequences

The most serious problems with passing all parameters by reference result from the assumption that all parameters can potentially be used for both input and output. This can be dangerous if the actual parameter is a variable since it means that an input variable may be inadvertently updated by a subprogram. This is called a *side effect* of the subprogram call. Although this may introduce a bug into the program that is difficult to find, it does not undermine the security of the FORTRAN system (i.e., it is not possible to corrupt the FORTRAN system's own data structures). This is not the case if a literal constant or expression is the actual parameter to the subprogram. Consider the following subroutine definition:

```
SUBROUTINE SWITCH (N)
N = 3
RETURN
END
```

`SWITCH` simply writes 3 into its parameter. Therefore, the invocation `CALL SWITCH(I)` would result in 3 being stored in `I`. Now consider the invocation `CALL SWITCH(2)`. What will be the effect of this? Recall that in our pseudo-code program (Chapter 1) we stored the constants to be used by the program in memory locations; for example, we stored the constant zero in location `000`. This is a very common practice, and compilers frequently allocate an area of memory, called a *literal table*, to contain the values of all of the literal constants used in a program. To return to our example, the literal 2 would be assigned a location in the literal table, and it is the address of this location that is passed to the subroutine. Therefore, when the subroutine assigns 3 to its parameter it will overwrite the value 2 in the literal table. The problem with this is that the compiler will compile a program so that it loads the contents of this location whenever it needs a constant 2, even though that location now contains a 3. Therefore, if the program now executed the assignment `I = 2 + 2`, the value stored in `I` would be 6. We have used the `SWITCH` procedure to change a "constant." You can imagine the debugging difficulties this could cause as the programmer tries to discover how the variable `I` has become set to 6 when the program clearly states `I = 2 + 2`.

You might object that this would never happen in practice, that a programmer would never write a subroutine like `SWITCH`. Unfortunately, there is a legend that a programmer once used exactly this device because he had uniformly used an incorrect constant throughout his program. Needless to say, this kind of practice results in unmaintainable programs. Even if this story is not true, there is a much more common cause of this situation. Both subroutines and functions often have output parameters, for example, to reflect the manner in which the subprogram completed its task. Programmers frequently are unaware that these parameters are used for output, either because the subprograms are poorly documented, or because the programmers have read the documentation and forgotten this detail. Thus, it is quite possible that the programmer could inadvertently pass a constant as an output parameter. This is an example of a lack of *security* in the language implementation, since the compiler allows its own runtime data structures (e.g., the literal table) to be corrupted. Recall

The Security Principle

No program that violates the definition of the language, or its own intended structure, should escape detection.

You might object that the compiler should not permit this; that since the actual parameter is a *constant* it cannot be stored into, that storing is only allowed for *variables*. You might also point out that when a constant or expression is used as an actual parameter, the programmer intends for the *value* of that actual parameter to be passed, not some address that the compiler happens to have stored that value into. These are both correct observations, and we will see that newer languages allow the programmer to distinguish between parameters that are intended to be used for input or output. The compiler can then ensure that a constant or expression is not used where an output parameter is expected. This also has an efficiency benefit, since it avoids needless indirect reference to input parameters.

Pass by Value-Result is Preferable

Although the best solution to this problem is to allow the programmer to specify whether a parameter is to be used for input or output, there is another way of implementing FORTRAN's parameters that does not present a security loophole; this is pass by *value-result* (also called *copy-restore*). The idea of this approach is that the *value* of the actual parameter is copied into the formal parameter at subprogram entry, and the *result*, or final value of the formal, is copied into the actual parameter at subprogram exit. Since both of these operations are done *by the caller*, the compiler can know to omit the second operation (copying out the result) if the actual was a constant or expression. Thus, the caller takes the result value only if the actual is something that can be meaningfully stored into, that is, a variable or array element in FORTRAN. The callee never has direct access to the caller's variables; it has access only to its formals. The caller is responsible for moving values to and from the formals. Note that pass by value-result does not prevent the programmer from accidently having one of the variables altered, but at least it preserves the security of the implementation.

▩ ***Exercise 2-14*:*** Determine how the FORTRAN compilers available to you implement parameters. Check the manuals and try some experiments (such as the `SWITCH` subroutine) to determine the parameter passing mode used. What do the ANS FORTRAN standards say about the way parameters are to be passed?

▩ ***Exercise 2-15:*** Consider the following FORTRAN `SUBROUTINE`:

```
SUBROUTINE TEST (X, Y, Z)
X = 1
Z = X + Y
RETURN
END
```

and consider the following code fragment:

```
N = 2
CALL TEST (N, N, M)
```

What will be the final value of `M` if the parameters are passed by reference? What will it be if they are passed by value-result?

■ ***Exercise 2-16*:*** There are several different varieties of pass by value-result: The address of the actual can be computed once, at subprogram entry time, or twice, once on entry and once on exit. Describe the output of this program under each of these two varieties of value-result:

```
DIMENSION A(2)
I = 1
A(1) = 10
A(2) = 11
CALL SUB (I, A(I))
PRINT, A(1), A(2)
END

SUBROUTINE SUB (K, X)
PRINT, X
K = 2
X = 20
RETURN
END
```

Does the outcome depend on the order in which the results are copied from the formals back into the actuals?

Subprograms Are Implemented Using Activation Records

In this section we investigate the way subprograms (subroutines and functions) are implemented. What happens when a subprogram is invoked? Clearly, we must transmit the parameters to the subprogram, which is the first step. This may be done by reference or by value (the first half of the value-result process).

It may seem that the next step is to enter the subprogram, but we must do something else first. If we entered now, there would be no way to get back to the caller because a subprogram can be called from many different callers and from many different places within one caller (as we saw in the `DIST` example). Therefore, there is not a unique place to which the callee should return when it has finished. To put it another way, it is necessary to tell the callee who its caller is; then, when the callee executes its `RETURN` statement it will know to whom to return.

There is one other issue we must address: saving the state of the caller. As you probably know, most computers have a number of *registers* that can be used for high-speed temporary storage. Since subprograms may be separately compiled, it is not usually possible to know the registers that another subprogram uses. Therefore, when one subprogram calls another, it is necessary for one or the other to preserve the content of the registers in a private area of memory. The content of the registers can then be restored when the caller gets control back from the callee.

We summarize these ideas as follows: When one subprogram calls another, the *state* of the caller must be preserved before the callee is entered and must be restored after the callee returns. By the state of the caller, we mean all of the information that characterizes the state of the computation in progress. This includes the contents of all of the variables, the contents of any registers in use, and the current point of execution (as indicated by the `IP`, or instruction pointer register). Of course, any information that is already stored in memory locations private to the caller is already saved and need not be saved again. All of the other information must be stored in a caller-private data area before the callee is entered. This data area is often called an *activation record* because it holds all the information relevant to one *activation* of a subprogram. A subprogram is *active* if it has been called but has not yet returned. In a nonrecursive language such as FORTRAN, there is one activation record for each subprogram (activation records for recursive subprograms are discussed in Chapter 3). The concept of an activation record is very important and will be discussed repeatedly in the following chapters.

The activation record serves a number of useful functions. For example, we have said that the caller must *transmit* the actual parameters to the callee. This means that the actual parameters (either their values or their references) must be placed in a location where the callee knows to find them. Where should this be? The callee's activation record is often a good choice (although there are others, such as registers). Thus, a subprogram's state includes its current parameters.

Since the activation record contains all of the information needed to restart a subprogram (its `IP`, register values, etc.), it is convenient to think of the activation record as a repository for all information relevant to the subprogram. We said earlier that the callee must be passed some sort of reference to the caller so that it will know which caller to resume when it returns. A very convenient way to do this is to transmit to the callee a pointer to the caller's activation record. The callee then has all of the information required to resume its caller. For example, the return address for the return jump to the caller can be obtained by the callee from the caller's activation record.

How are we to transmit to the callee the pointer to the caller's activation record? The simplest method is to store the pointer in the callee's activation record. This pointer from a callee's activation record to its caller's activation record is called a *dynamic link*. The *dynamic chain* is the sequence of dynamic links that reach back from each callee to its caller. The dynamic chain begins at the currently active subprogram (i.e., the one now in control) and terminates at the main program. See Figure 2.3 for an example. We show the situation in which the main program has called *S*, *S* has called *T*, *T* has called *F*, and *F* is still active.

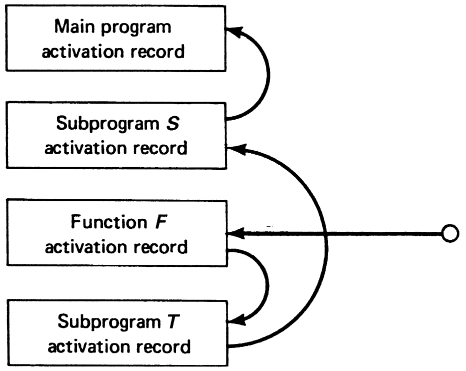

Main program activation record
Subprogram *S* activation record
Function *F* activation record
Subprogram *T* activation record

Figure 2.3 The Dynamic Chain of Activation Records

Let's summarize the tasks that must be completed to perform a subprogram invocation.

1. Place the parameters in the callee's activation record.
2. Save the state of the caller in the caller's activation record (including the point at which the caller is to resume execution).
3. Place a pointer to the caller's activation record in the callee's activation record.
4. Enter the callee at its first instruction.

The steps required to return from the callee to the caller are as follows:

1. Get the address at which the caller is to resume execution and transfer to that location.
2. When the caller regains control, it will have to restore the rest of the state of its execution (registers, etc.) from its activation record.

In addition, if the callee were a function (as opposed to a subroutine), then the value it returns must be made accessible to the caller. This can be accomplished by leaving it in a machine register or by placing it in a location in the caller's activation record.

From this discussion, we can see that an activation record must contain space for the following information:

1. The parameters passed to this subprogram when it was last called (we call this part `PAR`, for parameters)
2. The `IP`, or resumption address, of this subprogram when it is not executing
3. The dynamic link, or pointer to the activation record of the caller of this subprogram (we call this `DL`)
4. Temporary areas for storing register contents and other volatile information (we call this `TMP`)

It is not particularly important what format is used for this information, although certain layouts may be particularly efficient on certain machines (Figure 2.4).

We can make these ideas a little more specific by looking at the code for each of the steps in a `CALL` and a `RETURN`. To do this, we have to introduce some notation. Rather than introduce the assembly language for either a real or made-up machine, we use a conventional high-level language syntax. We must be careful to use statements that are very simple so that they can be implemented with one or two instructions on most machines. First, we use the notation `M[`*k*`]` to represent the memory location with the address *k*. For example,

```
M[R1] := IP;
```

• `PAR`	parameters
• `IP`	resumption address
• `DL`	dynamic link
• `TMP`	temporary storage

Figure 2.4 Components of Nonrecursive Activation Record

of the computation in progress. This includes the contents of all of the variables, the contents of any registers in use, and the current point of execution (as indicated by the `IP`, or instruction pointer register). Of course, any information that is already stored in memory locations private to the caller is already saved and need not be saved again. All of the other information must be stored in a caller-private data area before the callee is entered. This data area is often called an *activation record* because it holds all the information relevant to one *activation* of a subprogram. A subprogram is *active* if it has been called but has not yet returned. In a nonrecursive language such as FORTRAN, there is one activation record for each subprogram (activation records for recursive subprograms are discussed in Chapter 3). The concept of an activation record is very important and will be discussed repeatedly in the following chapters.

The activation record serves a number of useful functions. For example, we have said that the caller must *transmit* the actual parameters to the callee. This means that the actual parameters (either their values or their references) must be placed in a location where the callee knows to find them. Where should this be? The callee's activation record is often a good choice (although there are others, such as registers). Thus, a subprogram's state includes its current parameters.

Since the activation record contains all of the information needed to restart a subprogram (its `IP`, register values, etc.), it is convenient to think of the activation record as a repository for all information relevant to the subprogram. We said earlier that the callee must be passed some sort of reference to the caller so that it will know which caller to resume when it returns. A very convenient way to do this is to transmit to the callee a pointer to the caller's activation record. The callee then has all of the information required to resume its caller. For example, the return address for the return jump to the caller can be obtained by the callee from the caller's activation record.

How are we to transmit to the callee the pointer to the caller's activation record? The simplest method is to store the pointer in the callee's activation record. This pointer from a callee's activation record to its caller's activation record is called a *dynamic link*. The *dynamic chain* is the sequence of dynamic links that reach back from each callee to its caller. The dynamic chain begins at the currently active subprogram (i.e., the one now in control) and terminates at the main program. See Figure 2.3 for an example. We show the situation in which the main program has called *S*, *S* has called *T*, *T* has called *F*, and *F* is still active.

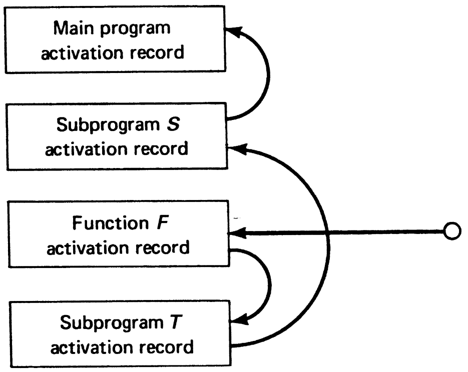


Figure 2.3 The Dynamic Chain of Activation Records

Let's summarize the tasks that must be completed to perform a subprogram invocation.

1. Place the parameters in the callee's activation record.
2. Save the state of the caller in the caller's activation record (including the point at which the caller is to resume execution).
3. Place a pointer to the caller's activation record in the callee's activation record.
4. Enter the callee at its first instruction.

The steps required to return from the callee to the caller are as follows:

1. Get the address at which the caller is to resume execution and transfer to that location.
2. When the caller regains control, it will have to restore the rest of the state of its execution (registers, etc.) from its activation record.

In addition, if the callee were a function (as opposed to a subroutine), then the value it returns must be made accessible to the caller. This can be accomplished by leaving it in a machine register or by placing it in a location in the caller's activation record.

From this discussion, we can see that an activation record must contain space for the following information:

1. The parameters passed to this subprogram when it was last called (we call this part `PAR`, for parameters)
2. The `IP`, or resumption address, of this subprogram when it is not executing
3. The dynamic link, or pointer to the activation record of the caller of this subprogram (we call this `DL`)
4. Temporary areas for storing register contents and other volatile information (we call this `TMP`)

It is not particularly important what format is used for this information, although certain layouts may be particularly efficient on certain machines (Figure 2.4).

We can make these ideas a little more specific by looking at the code for each of the steps in a `CALL` and a `RETURN`. To do this, we have to introduce some notation. Rather than introduce the assembly language for either a real or made-up machine, we use a conventional high-level language syntax. We must be careful to use statements that are very simple so that they can be implemented with one or two instructions on most machines. First, we use the notation `M[`*k*`]` to represent the memory location with the address *k*. For example,

```
M[R1]  :=  IP;
```

• `PAR`	parameters
• `IP`	resumption address
• `DL`	dynamic link
• `TMP`	temporary storage

Figure 2.4 Components of Nonrecursive Activation Record

could be an instruction to store the contents of the IP register in the memory location whose address is in the R1 register.

We also need a notation for denoting activation records and their parts. If *S* is a subprogram, then AR(*S*) is the address of its activation record. We also use a dot to select the *fields* of the activation record; for example,

```
M[AR(S)].DL
```

denotes the dynamic link field of *S*'s activation record.

We now look at the code in the caller for each part of a subprogram invocation. We assume that the caller, *S*, is going to invoke the function *F* (*P*, *Q*). The first step is to save the state of the caller, including the address at which it is to be resumed, in the caller's activation record. The resumption address is the address of the instruction after that which enters the callee; we call it R:

```
save registers etc. in M[AR(S)].TMP;

M[AR(S)].IP := R;
```

The second step is to compute the actual parameters *P* and *Q* and store their references in the PAR part of *F*'s activation record:

```
M[AR(F)].PAR[1]  :=  reference to P;
M[AR(F)].PAR[2]  :=  reference to Q;
```

The third step is to store a pointer to the caller's activation record, AR(*S*), into the DL field of the callee's activation record:

```
M[AR(F)].DL  :=  AR(S);
```

The final step in calling the function is to start execution at its entry address, which we will denote by entry(*F*):

```
goto entry(F);
```

When the callee returns, the caller begins executing at the next instruction, which we have called R. When the caller regains control, it will have to restore its registers from its activation record and, if the subprogram is a function, fetch the answer from wherever the callee left it (a register or the caller's activation record):

```
R:
  restore registers etc. from M[AR(S)].TMP;
  get value returned by F;
```

What code must the callee execute to return from a subprogram? As we saw before, the first step is to place the value to be returned in a place accessible to the caller. In FORTRAN the value returned is the last value assigned to a variable with the same name as the function, *F* in this case. Therefore, the answer is transmitted to the caller by placing the value of this variable in a machine register or in a field in the caller's activation record.

The pointer to the caller's activation record is just the dynamic link, that is, `M[AR(F)].DL`. The last step in a return is to transfer to the caller's resumption address; this is in the `IP` field of the caller's activation record:

```
goto M[M[AR(F)].DL].IP;
```

The code to invoke and return from a subprogram is summarized in the *translation rules* in Figures 2.5 and 2.6. Figure 2.5 shows the code produced in the caller for $F(P_1, \ldots, P_n)$; Figure 2.6 shows the code produced in a subroutine for a `RETURN`-statement.

▪ ***Exercise 2-17:*** The above implementation works for parameters passed by reference. What modifications would have to be made to accommodate pass by value-result?

▪ ***Exercise 2-18*:*** Design an activation record format appropriate for some machine with which you are familiar. Implement the code sequences in Figures 2.5 and 2.6 in the assembly language of that machine.

▪ ***Exercise 2-19*:*** The particular code sequences discussed above were chosen because they will generalize to other languages, for example, ones with recursive procedures. There are several improvements that can be made to the above code sequence, if we know that procedures are nonrecursive, as in FORTRAN. Find these improvements and estimate the savings (in terms of memory references per invocation) that would result.

Code in Caller S for $F(P_1, \ldots, P_n)$:

$F\ (P_1, \ldots, P_n) \Rightarrow$

```
     save registers etc. in M[AR(S)].TMP;
     M[AR(S)].IP := R;
     M[AR(F)].PAR[1] := reference to P1;
       .
       .
       .
     M[AR(F)].PAR[n] := reference to Pn;
     M[AR(F)].DL := AR(S);
     goto entry(F);
R:
     restore registers etc. from M[AR(S)].TMP;
     get value returned by F, if it's a function;
```

Figure 2.5 Implementation of Nonrecursive Call

Code in Callee *F* for `RETURN`:

```
RETURN ⇒
   place returned value where accessible to caller, if F a
     function;
   goto M[M[AR(F)].DL].IP;
```

Figure 2.6 Implementation of Nonrecursive Return

2.4 DESIGN: DATA STRUCTURES

Data Structures Were Suggested by Mathematics

FORTRAN is a scientific programming language. Therefore, the data structuring methods included in FORTRAN were those most familiar to scientific and engineering applications of mathematics: scalars and arrays. Languages from this period (ca. 1960) that were intended for commercial applications, such as COBOL and its predecessors (e.g., FLOW-MATIC, 1957), included data structuring methods appropriate to these applications, for example, character strings and records. As we will see, the data structuring methods provided by all these programming languages were also largely determined by the computer architectures and programming techniques of that time.

The Primitives Are Scalars

Since scientific programming makes heavy use of numbers, it follows that numeric scalars are the primary data structure *primitives* provided in FORTRAN. Most computers designed after the IBM 704 included both integer and floating-point arithmetic. As we saw in the pseudo-code of Chapter 1, integers are most commonly used for indexing and counting, and floating-point numbers are used for evaluating mathematical and physical formulas. In the early 1960s, a number of computers began to appear that supported *double-precision* arithmetic, that is, floating-point arithmetic with numbers twice as large as the normal floating-point numbers. To provide access to this, the FORTRAN II language included a *type* `DOUBLE PRECISION` and arithmetic operations on numbers of this type. FORTRAN II included several other important additions, including `COMPLEX` numbers, which are very important for scientific applications, and `LOGICAL` (or Boolean[6]) values. Next, we will discuss the primitive data types and the operations defined on them.

The Scalar Types Are Represented in Different Ways

Computers of the early 1960s were mostly *word-oriented*, that is, the basic addressable unit of storage was a *word*, which was also the basic unit in which most information was ma-

[6] George Boole, 1815–1864, was a British mathematician and logician whose work on formalized logic is widely used in computer science and electrical engineering.

nipulated. For example, integer, floating-point, and logical values all normally occupied one word. Even characters were normally manipulated in groups that corresponded to the number of characters that could fit in one word (typically about six).

Each of the primitive data types was *represented* in a manner appropriate to the operations defined on that data type. For example, integers were usually represented as a binary number with a sign bit:

s	b_{30}	b_{29}	$\cdots$	b_1	b_1	b_0

For the sake of this and the following examples, we will presume a computer with a 32-bit word. The number represented by this bit pattern is

$$(-1)^s \sum_{i=0}^{30} b_i 2^i$$

Of course, there are many possible ways to represent integers; for example, the sign bit could be at the other end of the word or a 1-bit could be used to represent positive numbers rather than negative numbers, as we have done. The operations provided by most computers for manipulating integers include the four arithmetic operations (addition, subtraction, multiplication, and division), tests for zero, and tests of the sign bit. The last can be used for greater-than and less-than comparisons by subtracting and testing the sign of the result. The primitive integer operations provided by FORTRAN were just those operations that could be implemented in one or two machine instructions; namely, the arithmetic operations, the comparisons, absolute value, and exponentiation to an integer power.

As you know, floating-point numbers are related to scientific notation, that is, to the convention of representing a number by a coefficient and a power of 10, for example, -1.5×10^3. A typical representation for a floating-point number is

sm	sc	c_7	$\cdots$	c_0	m_{21}	m_{20}	$\cdots$	m_1	m_0

The value represented by this number is $m \times 2^c$, where m is the *mantissa*,

$$m = (-1)^{sm} \sum_{i=0}^{21} m_i 2^{i-22}$$

and c is the *characteristic*,

$$c = (-1)^{sc} \sum_{i=0}^{7} c_i 2^i$$

Notice that the mantissa is always less than one. Again, there are many ways of representing floating-point numbers; the general idea is important here, not the details. The floating-point operations of FORTRAN follow closely those typically implemented on a computer: the four arithmetic operations, the comparisons, and absolute value. Exponentiation is also provided, although this is implemented through calls on library routines.

Notice that all of the numeric operations in FORTRAN are *representation independent*; that is, they depend on the logical, or *abstract*, properties of the data values and not on the details of their representation on a particular machine. A set of data values, together with a set of operations on those values that is defined without reference to the representation of the data values, is called an *abstract data type*. This is a very important concept that you will encounter many times in this book.

▨ ***Exercise 2-20*:*** A commonly cited advantage of FORTRAN is *portability*, that is, the ability to transport FORTRAN programs from one computer to another with few or no changes to the program. Discuss the relevance of abstract data types to portability.

The Arithmetic Operators Are Overloaded

In mathematics, the integers are taken to be a subset of the real numbers, and the real numbers are taken to be a subset of the complex numbers. The arithmetic operations (i.e., addition, subtraction, multiplication, and division) are so defined that the operations on reals are extensions of the operations on integers and the operations on complex numbers are extensions of the operations on reals. The result is that it is meaningful to write $x + y, x - y$, etc., regardless of whether x and y are integers, reals, or complex numbers. We can also see that it is perfectly reasonable to write $x + 1$, where x is a real number, since the integer 1 is also a real number. We have said that a primary goal of FORTRAN was to allow the programmer to write in he conventional algebraic notation; therefore, it was necessary for FORTRAN to support mixed expressions involving integer, floating-point, and complex numbers. This leads to a problem because computer numbers are not related in the same way as mathematical numbers; integers are not special kinds of floating-point numbers and floating-point numbers are not special kinds of complex numbers (we saw this in the last section). Indeed, the machine operations for performing integer arithmetic are completely different from those for performing floating-point arithmetic (this is necessary because the numbers are represented completely differently). Therefore, it is necessary to *overload* several meanings onto each arithmetic operation. For example, '+' can denote either integer, floating-point, or complex addition depending on its context, that is, depending on the type of the operands with which it is used. The compiler must look at this context in order to determine the machine instructions it must generate. This process can be expressed in rules such as these:

• Integer + integer	⇒ integer addition
• Real + real	⇒ floating-point addition
• Double precision + double precision	⇒ double floating addition
• Complex + complex	⇒ complex addition

There are several other terms you will hear in connection with overloaded operators. You will sometimes hear them called *generic* operators because they apply to a whole class of related data types (generic = relating to an entire group or class). The operations are also sometimes called *polymorphic* (*poly* = many, *morph* = form) because they have many code sequences corresponding to them.

As we said, it is common to mix operations on integers, reals, and complex numbers in mathematical formulas. Early FORTRAN systems did not permit this; for example, if a programmer wanted to add the real variable `X` to the integer variable `I`, it would have been necessary to *convert* `I` to floating-point, `X + FLOAT(I)`. Similarly, to assign `X` to `I` the programmer would write `I = IFIX(X)`. Later versions of FORTRAN allowed *mixed-mode expressions*, that is, expressions of more than one *mode*, or type. Such an expression is `I = X + I`, which is interpreted to mean

```
I = IFIX (X + FLOAT(I))
```

We say that `I` has been *coerced* from integer to real and that `X + I` has been coerced from real to integer. A *coercion* is an implicit, context-dependent type conversion. In this case, when an integer (e.g., `I`) appears in a context where a real is expected (e.g., addition to a real number), the integer is implicitly converted to a real. Most programming languages coerce integers to reals (this is required for a conventional algebraic notation), and some languages provided a much more extensive set of coercions (e.g., PL/I and Algol-68). The coercion rules of a programming language can be expressed in transformation rules that reflect the automatic insertion of the type conversion. For example, if we let `X` be a real variable or expression and `I` be an integer variable or expression, then the FORTRAN coercion rules are:

```
X + I  ⇒  X + FLOAT(I)
I + X  ⇒  FLOAT(I) + X

X - I  ⇒  X - FLOAT(I)
```

etc.

```
X = I  ⇒  X = FLOAT(I)
I = X  ⇒  I = IFIX(X)
```

Notice that although reals can be coerced to integers on assignment, in expressions integers can be coerced to reals but reals cannot be coerced to integers; reals are said to *dominate* integers. This results from the fact that, mathematically, integers are a subset of the reals. It should be noted that the 1996 ANS FORTRAN Standard does *not* permit integers and floating-point numbers to be mixed in one expression although it does permit mixed assignments and mixed expressions involving real, double-precision, and complex numbers. Newer versions of FORTRAN do permit mixed-mode expressions, but they sometimes give surprising results. For example, if we write `X**(1/3)` to compute the cube-root of a real variable `X`, we will get the wrong answer. This is because `1/3` is an integer division, which is truncating and returns 0; thus `X` is raised to the zeroth power. Therefore we must write `X**0.3333333333` or `X**(1.0/3)` to ensure the exponent is a real number. On the other hand, if we write `X**2.0` to square `X`, we will get the correct answer, but less efficiently than if we had written the mixed-mode expression `X**2`. This is because most compilers compute `X**2` the same as `X*X`, but implement `X**2.0` by subroutine calls on the log and antilog functions.

The Integer Type Is Overworked

In FORTRAN the integer type is required to do double duty: It represents both integers and character strings. This results from the fact that a *Hollerith*[7] *constant* is considered to be of

[7] This is named for Herman Hollerith who developed in the nineteenth century the 80-column punched card and the character code used on it.

type integer. A Hollerith constant was an early form of what is now called a *character string*; for example, the Hollerith constant `6HCARMEL` represents the six-character string 'CARMEL'. The syntax of a Hollerith constant is (1) a literal integer (e.g., '6'), followed by (2) the letter `H` (for "Hollerith"), followed by (3) the number of characters specified by the literal integer (e.g., the six characters 'CARMEL'). (Newer versions of FORTRAN permit familiar quoted character strings.)

Character strings are not *first-class citizens* in FORTRAN; this means that it is not possible to use them in all the ways we would normally expect to use data values. This violates the Regularity Principle. For example, there is no such thing as a Hollerith variable and there are no Hollerith comparison operations. Instead, FORTRAN permits character strings to be read into integer or real variables and permits Hollerith constants to be used as parameters where integers are expected. For example, given the function definition

```
FUNCTION ISUCC (N)
ISUCC = N + 1
RETURN
END
```

FORTRAN permits us to write

```
N = ISUCC (6HCARMEL)
```

even though this makes absolutely no sense. This is an example of *weak typing* since FORTRAN allows (in some circumstances) something of one type (e.g., Hollerith) to be used where another type (e.g., integer) is expected solely because they are represented the same way (a one-word bit-string). This is a *security* loophole because it allows meaningless operations to be performed without any warning, for example, adding 1 to a character string. This is a violation of the Security Principle since it allows a program that violates the definition of the language to escape detection.

FORTRAN's traditional lack of facilities for dealing with character strings has forced many programmers to write *machine-dependent* (and hence *nonportable*) programs that use integer and logical operations to manipulate characters. (What principle does this violate?) The FORTRAN 77 Standard has alleviated this problem by defining a `CHARACTER` data type with some rudimentary operations on fixed-length character strings; characters are no longer such second-class citizens.

The Data Constructor Is the Array

As we said before, the data structuring methods of FORTRAN are those of science and engineering—scalars and arrays. We have seen in the previous sections the various scalars provided by FORTRAN. We call these the *primitive* data structures because they are the basic data from which more complex data structures are built. The linguistic methods used to build complex data structures from the primitives are called the *constructors* for data structures. This terminology is used throughout this book to denote the two aspects of a structuring mechanism. In FORTRAN the only data structure constructor is the array. The remainder of this section is devoted to a discussion of arrays.

Arrays Are Static and Limited to Three Dimensions

In FORTRAN arrays are declared by means of a `DIMENSION` statement. For example,

```
DIMENSION DTA(100), COORD(10,10)
```

declares `DTA` to be a 100-element array (with subscripts in the range 1–100) and `COORD` to be a 10×10 array (with subscripts in the range 1–10). We can see that with the exception of a little "syntactic sugar," the first of these is really the same as the array declaration in our pseudo-code:

```
VAR DTA 100
+0000000000
```

Note that FORTRAN does not require arrays to be initialized.

FORTRAN is like the pseudo-code (and, incidentally, unlike most assembly languages) in requiring the dimensions of the array to be *integer denotations*, that is, literal integer constants (e.g., '100'), that are fixed for all time. This permits the FORTRAN system to allocate storage in the same simple way that our pseudo-code did. However it makes programs hard to maintain, since in a typical program many arrays, loops, etc. will have the same number. When the dimension of an array is changed, it will be difficult to tell which of these numbers need to be changed and which do not. (FORTRAN 77 permits named constants for array bounds and the like.)

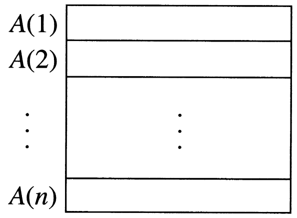
FORTRAN limits arrays to at most three dimensions (seven in FORTRAN 77) for efficiency reasons that are discussed later. In Chapter 3 we will see that this violates an important Regularity Principle (called the Zero-One-Infinity Principle) that tells us to avoid numbers like three and seven when we design languages.

Array Implementation Is Simple and Efficient

We saw in Chapter 1 that one-dimensional arrays are implemented as contiguous regions of memory; this permits indexing to be accomplished by adding the index value to the base address of the array. In this section we investigate the implementation of arrays in more detail. Consider first a one-dimensional array A declared by

DIMENSION $A(n)$

where n represents an integer denotation. The layout of A in memory is



If we let $\alpha\{A(1)\}$ mean the address in memory of $A(1)$, then we can see that the address of

$A(2)$ is $\alpha\{A(1)\} + 1$, the address of $A(3)$ is $\alpha\{A(1)\} + 2$, and so on, to the address of $A(n)$, which is $\alpha\{A(1)\} + n - 1$. In general, we have

$$\alpha\{A(i)\} = \alpha\{A(1)\} - 1 + i$$

This is called the *addressing equation* for the array A. (It assumes that each array element is stored in one addressable storage unit.)

Next, we derive the addressing equation for a two-dimensional array. Suppose A is declared by

DIMENSION $A(m, n)$

FORTRAN arranges arrays in memory in *column-major* order, that is, with the columns occupying adjacent memory locations as shown in Figure 2.7. (Most programming languages either use *row-major* order or, more frequently, encourage machine independence by not specifying the layout at all.) If we look at Figure 2.7 for a pattern, we can see that the address of $A(I, J)$ is $\alpha + (J - 1)m + I - 1$; therefore, the addressing equation for two-dimensional column-major arrays is

$$\alpha\{A(I,J)\} = \alpha\{A(1,1)\} + (J - 1)m + I - 1$$

- ***Exercise 2-21:*** Test the above equation by computing the address of $A(1,1)$, $A(m,1)$, $A(1,2)$, $A(m,2)$, $A(1,3)$, and $A(m,n)$ and comparing the results with Figure 2.7.

- ***Exercise 2-22:*** Derive the addressing equation for two-dimensional arrays stored in *row-major* order.

- ***Exercise 2-23:*** Derive the addressing equation for three-dimensional arrays stored in column-major order; do the same for row-major order.

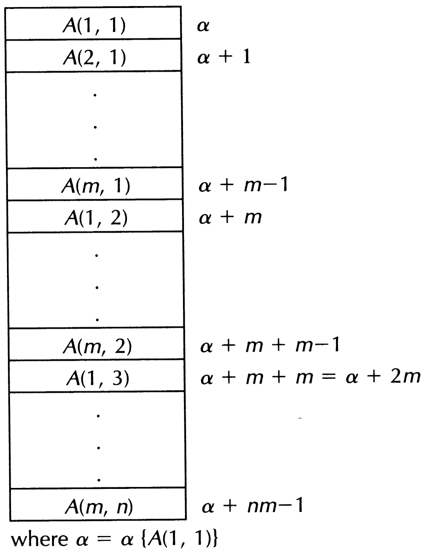


Figure 2.7 Column-Major Layout of Two-Dimensional Array

▩ ***Exercise 2-24:*** Double-precision and complex numbers both occupy two words each rather than just one. Derive the addressing equation for one-, two-, and three-dimensional arrays of double-precision or complex numbers.

FORTRAN Arrays Allow Many Optimizations

We said in the beginning of this chapter that efficiency was the primary goal of the FORTRAN system. We also saw that the FORTRAN system was introduced simultaneously with the inclusion of index registers in computer hardware. It was thus crucial that FORTRAN make optimum use of the index registers. Let's look at a concrete case, a simple `DO`-loop.

```
      DO 20 I = 1, 100
      SUM = SUM + A(I)
20    CONTINUE
```

The obvious implementation of this loop is to initialize to 1 an index register (corresponding to `I`) at the beginning of the loop and to increment the index register on each successive iteration. The reference to `A(I)` in the body of the loop would be implemented in the straightforward way: computing the address of the array element according to the addressing equation $\alpha\{(A(1)\} - 1 +$ `IR` (we have used `IR` to represent the contents of the index register). Thus, on each iteration of the loop an addition must be performed adding the constant $\alpha\{A(1)\} - 1$ to the contents of the index register `IR`. The loop can be summarized by this code:

```
        initialize IR to 1;
loop:   compute the address α{A(1)} - 1 + IR;
        fetch the contents of this and add to SUM;
        increment IR;
        if IR is less than or equal to 100, goto loop.
```

The problem is that this addition is unnecessary. A smart assembly language programmer could avoid it by initializing the index register to the address of the first element of the array and then using the index register as an indirect address for the array element. That is, the loop would be implemented:

```
        initialize IR to α{A(1)};
loop:   fetch the contents of the location whose address
          is in IR and add to SUM;
        increment IR;
        if IR is less than or equal to α{A(100)}, goto loop.
```

We can see that this saves an addition on each iteration; it was optimizations such as this that sold the first FORTRAN compiler.

You are probably aware that most languages allow subscripts that are more complicated than simple variables, for example, `A(I + 1)`, `A(3*I - 6)`, and `A(1 + A(J))`. You can probably also see that elaborate expressions such as these could so complicate analysis that it would make optimizations of the type described above impossible. For this reason early versions of FORTRAN restricted subscript expressions to one of the following forms:

c
v
$v + c$ or $v - c$
$c*v$
$c*v + c'$ or $c*v - c'$

where c and c' are integer denotations, and v is an integer variable. Therefore, `A(2)`, `A(I)`, `A(I - 1)`, and `A(2*I - 1)` are all legal array references, but `A(1 + I)`, `A(I - J)`, `A(100 - 1)`, and `A(I*J)` are not. Of course, this seemingly arbitrary restriction is very confusing for programmers. Why was it adopted? The reason is that it allows just the sort of optimizations we have been discussing. Consider the most complicated case, an array reference of the form `A(`c`*I + `f`)`, where c and f are integer denotations and `I` is the controlled variable of a `DO`-loop. To perform the optimization shown above, we want to factor out all of the addressing equation that does not vary from one iteration to the next. This is called "removing invariant code from the body of a loop." To see what the variant part is, let's compute the difference between the I-th and $(I + 1)$-st iterations:

$$\begin{aligned}&\alpha\{A(c*(I + 1) + f)\} - \alpha\{A(c*I + f)\}\\ &= [\alpha\{A(1)\} + c(I + 1) + f - 1] - [\alpha\{A(1)\} + cI + f - 1]\\ &= c(I + 1) - cI = c\end{aligned}$$

Therefore, the address changes by c on each iteration, so if the address of the current array element is kept in an index register, all that is necessary is to add c to that index register on each iteration. To what value should the index register be initialized? If we suppose the loop begins with `I` $= 1$, we can compute

$$\alpha\{A(c*I + f)\} = \alpha\{A(c + f)\} = \alpha\{A(1)\} + c + f - 1$$

which, since c and f are constants, is itself a constant.

■ ***Exercise 2-25:*** Show that the same optimizations will not work for array subscripts of the form $u*v$, where u and v are variables.

We now investigate two-dimensional arrays; that is, we will optimize the use of index registers for two nested loops. We take as an example:

```
      DO 20 I = 1, 100
         DO 30 J = 1, 64
            SUM = SUM + A(I,J)
30       CONTINUE
20    CONTINUE
```

The analysis will be performed for simple variable subscripts; you will extend it in an exercise. What is the difference in array element addresses for each iteration of the inner loop? Since this increments J, it is (assuming A is dimensional $m \times n$):

$$\begin{aligned}&\alpha\{A(I, J+1)\} - \alpha\{A(I, J)\}\\ &\quad = [\alpha\{A(1,1)\} + (J+1-1)m + I - 1] - [\alpha\{A(1,1)\} + (J-1)m + I - 1]\\ &\quad = Jm - (J-1)m = m\end{aligned}$$

Thus, the index register must be incremented by m on each of the inner iterations. On each of the outer iterations, I is advanced so the index register must be incremented by

$$\begin{aligned}&\alpha\{A(I+1, J)\} - \alpha\{A(I, J)\}\\ &\quad = [\alpha\{A(1,1)\} + (J-1)m + (I+1) - 1] - [\alpha\{A(1,1)\} + (J-1)m + I - 1]\\ &\quad = 1\end{aligned}$$

Therefore, in the outer loop the index register is incremented by 1 on each iteration.

▩ ***Exercise 2-26:*** Perform an analysis analogous to that above, but assume that the loops are nested so that the first subscript varies more rapidly than the second.

▩ ***Exercise 2-27:*** Perform the above index register optimization analysis for the general two-dimensional subscript case, $A(c*I + f, d*J + g)$. Assume A is dimensioned $m \times n$.

▩ ***Exercise 2-28*:*** Perform the above analysis for three-dimensional arrays.

We can see from this last exercise why FORTRAN limited arrays to three dimensions. Backus has said that the number of special cases they had to consider increased exponentially with the number of subscripts, therefore, they chose three as a limit.

2.5 DESIGN: NAME STRUCTURES

The Primitives Bind Names to Objects

We pointed out in Section 2.4 that composite structures are built up by applying a set of *constructors* to the *primitive* structures. We will see that this is also the case with name structures, which are discussed in this section. One of the goals of this section is to make the notion of *name structure* clear. We can get a handle on name structures by considering the structures that we have already seen. What does a data structure structure? The data, of course. In other words, the purpose of a *data structure* is to organize the primitive data in order to simplify its manipulation by the program. What about a control structure? It's easy to see that the purpose of a *control structure* is to organize the control-flow of a program. Now, what can we say about name structures? By analogy we would expect them to organize the names that appear in a program, and this is correct. The subject of the remainder of this section (and of many other sections in this book) is the meaning of organizing names and what this organization accomplishes.

What are the primitive name structures of FORTRAN? These are simply those constructs that give meanings to names, that is, declarations or *binding constructs* (sometimes abbreviated *bindings*). In Chapter 1 we discussed the very simple means provided by the pseudo-code for binding symbolic names to addresses of memory locations. For example,

```
VAR SUM 1
+0000000000
```

bound the *identifier* SUM to a location in data memory (it happened to be location 002 in the example in Chapter 1) and initialized that location to zero. The pseudo-code's bindings have three functions: allocating a region of memory (one word long in this case), initializing that region (to zero in this case), and binding the identifier (SUM in this case) to the address of that region. These functions are performed by FORTRAN declarations, although in a slightly different form. For example,

```
INTEGER I, J, K
```

declares I, J, and K to be integer variables. This means that storage must be allocated for these variables (one word each), and the names I, J, and K must be bound to the addresses of these locations. Notice that this declaration does not perform the initialization function; this is accomplished with a separate DATA-statement in FORTRAN. The other important function fulfilled by this declaration is to specify the *type* of the variables, namely, INTEGER. Therefore, when these variables appear in an expression, the compiler will be able to determine which kind of operations to perform (e.g., integer addition or floating-point addition) and whether any coercions are necessary.

Declarations Are Nonexecutable

FORTRAN declarations are often called *nonexecutable* statements to differentiate them from *executable* statements such as assignment statements, GOTOs, IF-statements, and subroutine calls. Nonexecutable statements provide information for the compiler and other preexecution processors of the program (such as the linker and loader). For example, the type in the declaration (e.g., INTEGER) is used to determine the amount of storage that must be allocated for the variables. The compiler keeps track of the locations that have been allocated just as our pseudo-code loader did. Since this allocation is done once, before the program is executed, and never changes, it is called *static allocation*. As we will see in the following chapters, most programming languages now do *dynamic allocation*, that is, storage is allocated and recycled dynamically (i.e., at run-time) during program execution. Dynamic allocation is discussed at length in later chapters.

The pseudo-code interpreter kept track of the location of variables by placing them in a *symbol table*; compilers do exactly the same thing. For example, after a FORTRAN compiler has decoded the declaration 'INTEGER I, J, K', it will make entries in the symbol table for each of I, J, and K. These entries will contain the location of these variables in memory and their types, in other words, everything the compiler needs to know about these variables in order to generate correct code. We can visualize a symbol table like this:

Name	Type	Location
⋮	⋮	⋮
I	INTEGER	0245
J	INTEGER	0246
K	INTEGER	0247
⋮	⋮	⋮

Optional Variable Declarations Are Dangerous

FORTRAN has an unusual convention that has been abandoned in almost all newer programming languages: automatic declaration of variables. In other words, if a programmer uses a variable but never declares it, the declaration will be automatically supplied by the compiler. This was originally conceived as a convenience for programmers since it saved them the effort of declaring all of their variables. As we will see below, it is a false economy since it undermines security.

You may ask, "If FORTRAN automatically declares undeclared variables, then how does it know what type the variable should be declared to be?" In other words, if the statement `I = I + 1` appears in a program, and `I` has not been declared, then does the compiler declare it `INTEGER`, `REAL`, `COMPLEX`, or what? FORTRAN solves this problem with the "I Through N Rule," which states that any variables whose names begin with I through N are declared integers and all others are declared reals. This tends to correspond to mathematical convention since the variables *i*, *j*, *k*, *l*, *m*, and *n* are usually used for indexes and counts (i.e., integers), and the variables *x*, *y*, *z*, *a*, *b*, *c*, and so forth, are usually used for other things. This rule has the unfortunate consequence that it leads programmers to pick obscure names for variables (e.g., `KOUNT`, `ISUM`, `XLENGTH`) so that they don't have to declare them. Although FORTRAN does not dictate this style, it inclines programmers toward it.

There is a much more serious problem with this automatic declaration convention, however. Suppose a programmer intended to type the statement `COUNT = COUNT + 1` but accidently typed `COUNT = COUMT + 1` (i.e., an `M` instead of an `N`). What would be its effect? Since the variable `COUMT` has not been declared, the compiler will automatically declare it, as a real in this case. Since there is no `DATA`-statement to initialize it, it will probably contain whatever was left over in that memory location. The result will be that a strange and inexplicable value will be stored into `COUNT` and that the programmer will have a difficult debugging problem. If FORTRAN did not automatically declare variables, this error would have been caught at compile-time, since `COUMT` would have been reported as an undeclared variable (Security Principle). This is one example of how a small design error, such as a misplaced concern for writing convenience, can seriously undermine the security of a language and cause debugging nightmares for the programmer. We will see in a later section that this particular error also interacts with other small errors in the FORTRAN design to create other serious problems.

Environments Determine Meanings

As you are aware, the meaning of a sentence often depends on the *context* in which it is stated; words can have different meanings, depending on their context. The same is true in programming languages; a statement such as `X = COUNT(I)` may have many meanings, depending on the definitions of its constituent names. For example, `X` may be either an integer or a real and `COUNT` may be either an array or a function. The *context* of this statement, or any programming language construct, is thus the set of definitions that is *visible* to that statement or construct. Since the context of a construct can be thought of as surrounding that construct, a context is often called an *environment*. Thus, we can say that an environment determines the visibility of bindings. The concept of context or environment is very important in programming languages and is discussed many times in this book. In fact, name structures can be defined as the means of organizing the environment.

Variable Names Are Local in Scope

We saw earlier that FORTRAN programs are divided into a number of disjoint *subprograms* and that this division allows subprograms to be developed independently and stored in libraries. Suppose someone has defined a FORTRAN function `SUM` that sums the elements of an array. It really makes no difference to a user of this function whether the formal parameters are called `A` and `N`, or `X` and `SIZE`, or whatever. All a user of this function needs to know is that the first parameter is the array and the second parameter is its size. Similarly, the user does not care whether the programmer of the function used `I` for the controlled variable of the `DO`-loop, or `J`, or whatever. These are details of the implementation that are of no concern to the user of the function and, in fact, that may be changed at some point by the implementor of the function. Therefore, information such as this (e.g., the variable names used by the implementation of the function) should be hidden from the user, a principle called *Information Hiding*, which is discussed later (Chapter 7). Suffice it to say now that observing this principle makes programs much more maintainable.

Information hiding relates to environments in a very straightforward way: We have said that the caller of a subprogram should not be able to see the names of any variables declared in that subprogram (including the formal parameters). This means that these names are not *visible* to the caller, that is, they are not in the caller's environment. Therefore, since a FORTRAN program is made up of a number of disjoint subprograms, we can see that each of these subprograms must have a *disjoint environment* containing just the variables (and formal parameters) declared in that subprogram. This avoids cluttering the name space with a lot of variables that do not have to be accessible.

Subprogram Names Are Global in Scope

Clearly, the above argument cannot apply to the names of the subprograms themselves; if the name of a subprogram were visible only within that subprogram, then it would be impossible for anyone else to call that subprogram. Instead, FORTRAN specifies that subprogram names are visible throughout the program; they are thus said to be *global*. In contrast we say that variable names are *local* to the subprograms in which they are declared. Thus, the names fall into two broad classes, depending on their visibility. This property is called the *scope* of a name; that is, subprogram names have global scope and variable names have local scope. The *scope* of a binding of a name is defined as that region of the program over which that binding is visible. Consider Figure 2.8. In this program the scope of the binding of X marked (*) is the main program and the scope of the binding of X marked (**) is the subroutine R. Similarly, the scope of the binding of Y (**) is R and the scope of the binding (***) is S. The scopes of the two uses of N as formal parameters are the bodies of the corresponding subroutines. Finally, the scope of the declarations of the subroutines R and S is the entire program.

These relationships can be depicted in a *contour diagram*, such as that in Figure 2.9. These diagrams (which were invented by John B. Johnston[8]) should be interpreted as though the boxes were made of one-way mirrors that allow us to look out of a box but not into one. The arrows on the diagram show, for example, that R can see the bindings of S and its own Y, but that it cannot see the Y in S or the X in the main program. Contour diagrams are a valuable aid to visualizing scopes, particularly in languages with more complicated scope rules than FORTRAN. We will use them again in later chapters.

```
C     MAIN PROGRAM
      INTEGER X                (*)
       .
       .
       .
      CALL R(2)
       .
       .
       .
      CALL S(X)
       .
       .
       .
      END
      SUBROUTINE R(N)
      REAL X, Y                (**)
       .
       .
       .
      CALL S(Y)
       .
       .
       .
      END
      SUBROUTINE S(N)
      INTEGER Y                (***)
       .
       .
       .
      END
```

Figure 2.8 Examples of Variable Scopes in FORTRAN

[8] See Johnston (1971).

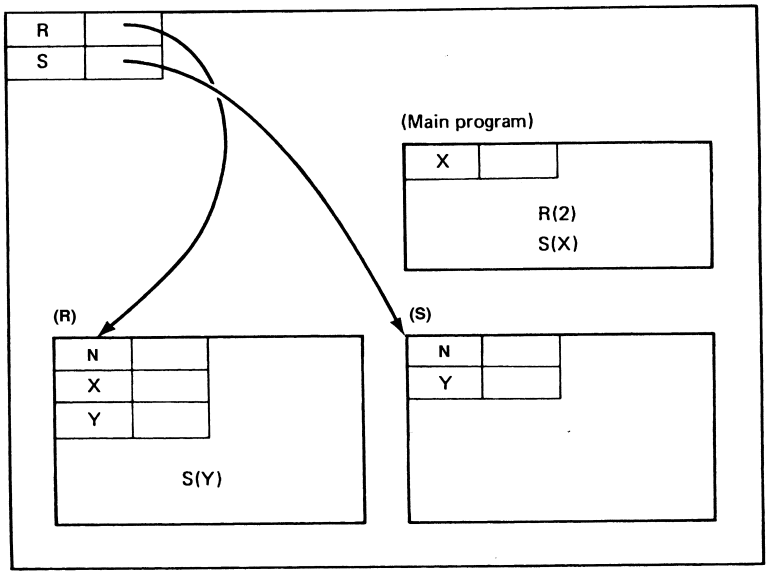


Figure 2.9 Contour Diagram Showing Scopes of Variables

▨ ***Exercise 2-29:*** Write in skeleton form a FORTRAN program involving at least three subprograms. Draw a contour diagram that shows the visibility relations between the bindings and uses of names.

It Is Difficult to Share Data with Just Parameters

Since all variables are subprogram local, it would seem that the only way to pass information from one subprogram to another is through parameters. Let's see what the consequences of this would be. Suppose that we need to implement a symbol table in FORTRAN; the simplest way to do this is to represent the symbol table as four parallel arrays. That is, each element of the array `NAMES` contains the name of a symbol, the corresponding element of the array `LOC` contains the location of the name, the array `TYPE` contains an integer code specifying the type of the name (integer, real, etc.), and the array `DIMS` contains the dimensions of those symbols that name arrays. We also need a number of subroutines and functions for managing this symbol table. For example, we need a `LOOKUP` function that takes a name (encoded as an integer) and finds the corresponding entry in the `NAMES` array. Also, we need subroutines that create new entries in the symbol table by entering the names, locations, types, and dimensions in the appropriate arrays. For example, we may want `VAR` that enters a simple variable and `ARRAY1` that enters a one-dimensional array.

Let's consider the organization of the name space of this program. The various arrays representing the symbol table constitute a database that is managed by the *accessing* subprograms, `LOOKUP`, `VAR`, `ARRAY1`, etc. The scope rules of FORTRAN permit a subprogram to access data from only two sources: the variables declared within the subprogram and the variables passed as parameters. Since the arrays representing the symbol table must be accessible to all of the accessing subprograms, these arrays cannot be local variables of any one of these subprograms. The only alternative is to declare these arrays in the main program and pass them explicitly to each of these subprograms. This means that each of these accessing subprograms must have four extra parameters that are the four arrays representing the symbol table. For example, to enter a declaration for an array whose name is in `NM`, with location `AVAIL`, type code `INTCOD`, and dimensions `M` and `N`, we write:

```
CALL ARRAY2 (NM, AVAIL, INTCOD, M, N, NAMES, LOC, TYPE, DIMS)
```

This is not very readable; the four extra parameters clutter up the `CALL`. In fact, the user of these symbol table subprograms really does not care whether the symbol table is represented by four arrays, or one array, or 20. On the contrary, requiring the user to supply this information causes a serious maintenance problem. Since the user of this symbol table package is required to declare the four arrays in the main program and pass them on each call of the accessing subprograms, it means that it will be very difficult for the implementor of these routines to change the organization of the symbol table. For example, if we wanted to modify it so that five arrays were required, we would have to track down every subroutine using these routines so that the appropriate changes could be made in the user's program. This could be very complicated since, if the user has subprograms that call the symbol table routines, these subprograms will have to have these same four array parameters. In other words, the details of the organization of the symbol table are scattered throughout the user's program. This means that even the slightest change in the symbol table's organization will have a devastating effect. The practical result will probably be that the symbol table package is never altered.

These observations are summarized in the Information Hiding Principle of D. L. Parnas.

Information Hiding Principle

Modules should be designed so that (1) The user has all the information needed to use the module correctly, *and nothing more*. (2) The implementor has all the information needed to implement the module correctly, *and nothing more*.

This principle is discussed in detail in Chapter 7.

COMMON Blocks Allow Sharing between Subprograms

For the reasons discussed above, using parameters to share data is unsatisfactory, so it is fortunate that FORTRAN provides a mechanism to avoid it. Let's be clear about the problem. What we are trying to do is provide a set of subprograms that collectively manages a shared database; this is a very common situation. The Information Hiding Principle suggests that the problems result from the fact that information about the representation of the database, which is relevant only to the accessing subprograms, has been spread to other parts of the program. This was necessitated by the scope rules of FORTRAN. What is needed is a mechanism that allows this information to be accessible to just the symbol table subprograms without being passed as explicit parameters. This mechanism is found in the FORTRAN `COMMON` block.

As suggested by their name, COMMON blocks permit subprograms to have common variables, that is, to share data areas (Figure 2.10). This can most easily be explained by seeing its application to our symbol table example. Since we want the accessing subprograms to share the four arrays representing the symbol table, we can place them in a COMMON block called SYMTAB that is declared in each of these subprograms. This is shown in outline form in Figure 2.11.

Only the subprograms that include a declaration of the COMMON block SYMTAB will have access to this data area. Other subprograms can share other data areas through COMMON blocks with different names.

COMMON Permits Aliasing, Which Is Dangerous

In the above example, we can see that each subprogram includes an identical specification of the SYMTAB COMMON block. This should lead us to ask the question, "What if all the specifications don't agree?" You might expect the compiler to diagnose this as an error, but this is not the case; indeed, FORTRAN does not require the programmer to use the same names in different specifications of the same COMMON block. For example, the VAR subroutine could specify this COMMON block by

```
COMMON /SYMTAB/ NM(100), WHERE(100), MODE(100), SIZE(100)
```

Then, VAR's use of WHERE(I) would be equivalent to ARRAY2's use of LOC(I). There's little point in doing this. Unfortunately, this could occur accidently; for instance, a programmer might reverse the LOC and TYPE arrays in one of the subprograms:

```
COMMON /SYMTAB/ NAMES(100), TYPE(100), LOC(100), DIMS(100)
```

Whenever this subprogram makes a change to what seems to be the LOC array, it will in fact be changing the TYPE array. Needless to say, bugs like this can be very difficult to find.

The problem is even more serious, however, because FORTRAN does not even require separate specifications of the same COMMON block to agree in *structure*. For example, one subprogram may contain the specification

```
COMMON /B/ M, A(100)
```

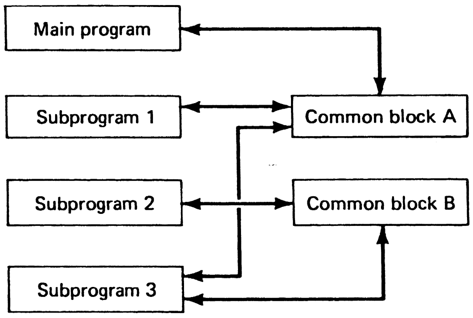


Figure 2.10 Subprogram Communication Through COMMON Blocks

```
C     MAIN PROGRAM
        .
        .
        .
      CALL ARRAY2 (NM, AVAIL, INTCOD, M, N)
        .
        .
        .
      END
      SUBROUTINE ARRAY2 (N, L, C, D1, D2)
      COMMON /SYMTAB/ NAMES(100), LOC(100), TYPE(100), DIMS (100)
        .
        .
        .
      END
      SUBROUTINE VAR (N, L, C)
      COMMON /SYMTAB/ NAMES(100), LOC(100), TYPE(100), DIMS(100)
        .
        .
        .
      END
      etc.
```

Figure 2.11 Example of the Use of COMMON Blocks

and another may contain

```
COMMON /B/ X, K, C(50), D(50)
```

The names included in each COMMON specification are made to correspond to locations in the block according to their position in the specification; this is shown in Figure 2.12.

We see that the location called M in the first subprogram is called X in the second program. These two variables don't even have the same type! If the programmer stores an integer in M and then uses X as a floating-point number, the result will be gibberish since integers and floating-point numbers are represented completely differently. This is a security loophole since it violates FORTRAN's type system; that is, it allows meaningless operations to be performed (floating-point operations on integer data). We also see that the array A par-

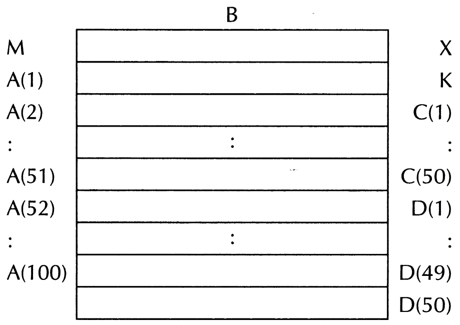


Figure 2.12 Aliasing in COMMON Blocks

tially overlaps the arrays C and D so that A(2) is the same location as C(1) and A(52) is the same location as D(1). Location D(50) is not even accessible from the first subprogram.

You might argue that this is a willfully pathological example and that no programmer would ever do this. In fact, programmers sometimes do exactly this to conserve storage (by making several arrays occupy the same storage locations) or to circumvent the type system on purpose.

They may also do these things accidently; for example, the specification of SYMTAB could be easily mistyped as

```
COMMON /SYMTAB/ NAMES(10), LOC(100), TYPE(100), DIMS(100)
```

This situation can also arise when a programmer updates a COMMON specification in one subprogram but forgets to update it in another.

In all of these cases, the problem is that FORTRAN COMMON blocks permit *aliasing*, the ability to have more than one name for the same memory location. Aliasing makes programs difficult to understand and difficult to maintain. We will discuss aliasing and means for avoiding it repeatedly in this book. We will see in later chapters that other languages have provided better solutions than COMMON blocks to the problem of shared data.

▩ ***Exercise 2-30:*** Show the correspondence of the two declarations of SYMTAB in the case mentioned above, in which NAMES(100) was mistyped as NAMES(10).

▩ ***Exercise 2-31:*** How does the Abstraction Principle apply to FORTRAN COMMON blocks?

▩ ***Exercise 2-32:*** Suggest some alterations to the FORTRAN COMMON facility that would make it more secure and safer to use.

EQUIVALENCE Allows Sharing within Subprograms

Recall that FORTRAN was developed at a time when computer memories were extremely small, often only a few thousand words. It, therefore, was mandatory that programs make optimum use of memory, for example, by sharing. In most large programs, each data structure will go through periods when it is being used and periods when it is not. For example, a large array may be used for holding data when it's first read in and for its initial processing. After this first stage, that array may no longer be needed, but a different array may be needed, for example, for the results of the computation. If the program is finished with the first array before it needs to begin using the second array, then memory will be better utilized if the two arrays can occupy the same area in memory. In FORTRAN this can be accomplished with an EQUIVALENCE declaration such as

```
DIMENSION INDATA(10000), RESULT(8000)
EQUIVALENCE (INDATA(1), RESULT(1))
```

This states that the first element of INDATA is to occupy the same memory location as the first element of RESULT. In other words, we have *explicitly aliased* the two arrays so that they will share storage. Needless to say, this has all of the problems of aliasing that we discussed in connection with COMMON. Although it was introduced as a method for economizing storage, it can be used for many purposes, including subversion of the type system. For example, a logical variable may be equivalenced to a real variable so that the programmer can use logical operations ('.AND.', '.OR.', '.NOT.') to access the parts of the floating-point representation of the number. This kind of programming leads to very machine-dependent, and hence nonportable, programs and is usually unnecessary anyway since there are better solutions to these problems. In many cases, these problems result from trying to use FORTRAN for applications for which it was not intended, such as string manipulation and systems programming.

The EQUIVALENCE statement has outlived its usefulness for a number of reasons. First, computer memories are now much larger so that economization of storage is no longer the problem that it was in the 1950s. Second, many computers now have *virtual memory* systems that automatically and safely manage the sharing of real memory. Finally, as we will see in later chapters, modern programming languages provide dynamic storage management facilities that also automatically and safely manage the allocation of memory.

■ ***Exercise 2-33:*** Suppose we have three arrays dimensioned A(1000), B(700), and C(700). Further suppose that A is never in use at the same time as B or C, but that B and C may be in use at the same time. Write an EQUIVALENCE statement to share storage as much as possible. How would you change your solution if B and C were never in use at the same time?

■ ***Exercise 2-34*:*** Most FORTRAN systems allow the same variables to appear in both COMMON and EQUIVALENCE statements. For example,

```
COMMON /B/ U(100), V(100)
DIMENSION W(100)
EQUIVALENCE (U(60), W(1)), (W(80), V(10))
```

What do you suppose the effects of these declarations are? You can consult a FORTRAN manual or try to figure it out on your own. What *should* be the effects of these statements?

2.6 DESIGN: SYNTACTIC STRUCTURES

Languages Are Defined by Lexics and Syntax

The *syntax* of a language is the way that words and symbols are combined to form the statements and expressions. In the previous sections, we indirectly discussed the syntax of many FORTRAN constructs. For example, we saw that a DO-loop has the word DO, followed by a statement number, followed by an integer variable, followed by the symbol '=', followed by an integer expression, followed by a comma symbol, followed by another integer expression, and optionally followed by another comma symbol and another integer expression. You have probably seen the syntax of programming languages described with a formal grammatical notation, for example,

```
⟨DO-loop⟩ ::= DO ⟨label⟩ ⟨var⟩ = ⟨exp⟩, ⟨exp⟩ [, ⟨exp⟩]
```

We will discuss these notations in Chapter 4.

The *lexics* of a language is the way in which characters (i.e., letters, digits, and other signs) are combined to form words and symbols. For example, the lexical rules of FORTRAN state that an identifier must begin with a letter and be followed by no more than five letters and digits. Lexical rules are also frequently expressed in formal grammatical notations; in fact, the term *syntax* is often used to refer to both the lexical and syntactic rules of a language. The *syntactic analysis* phase of a compiler is often broken down into two parts: the *lexical analyzer* (also called a *scanner*) and the *syntactic analyzer proper* (also called a *parser*). In the following sections, we will discuss the lexics and syntax of FORTRAN.

A Fixed Format Lexics Was Inherited from the Pseudo-Codes

The pseudo-code we developed in Chapter 1 was typical of these languages in its *fixed format* lexical conventions. It was card-oriented, that is, there was one instruction per card and particular columns of these cards were dedicated to particular purposes, for example, columns 1–4 for the operation. FORTRAN has similar lexical conventions, namely, one statement per card and columns dedicated to particular purposes:

Columns	Purpose
1–5	statement number
6	continuation
7–72	statement
73–80	sequence number

Since a statement may not entirely fit on one card, it can be continued onto following cards, if these continuation cards have a character punched in column 6. The bulk of the card, columns 7–72, was devoted to the actual statement, which was *free format*; that is, it did not have to be punched in specific columns. This general lexical style was common in languages of the late 1950s and early 1960s such as COBOL. Although it is adequate for use with punched card equipment, it is quite awkward for use with the interactive program preparation methods now generally in use. It also has other limitations that we will discuss later.

Ignoring Blanks Everywhere Is a Mistake

FORTRAN adopted the unfortunate lexical convention that blanks are ignored everywhere in the body of the statement. While this was certainly an improvement over the fixed fields of the pseudo-code interpreters, it was a significant deviation from most natural languages, in which blanks are significant.

Itisveryhardtoreadasentencewithnoblanks,

yet this is exactly what FORTRAN compilers were required to do. In FORTRAN, the statement

```
    DIMENSION IN DATA (10000), RESULT (8000)
```

is exactly equivalent to

```
DIMENSIONINDATA(10000),RESULT(8000)
```

and, for that matter,

```
D I M E N S I O N IN DATA (10 000), RESULT (8 000)
```

While this may seem to be a harmless convenience, in fact it can cause serious problems for both compilers and human readers. Consider this legal FORTRAN statement:

```
DO 20 I = 1. 100
```

which looks remarkably like the DO-statement:

```
DO 20 I = 1, 100
```

In fact, it is an assignment statement of the number 1.100 to a variable called DO201, which we can see by rearranging the blanks:

```
DO20I = 1.100
```

You will probably say that no programmer would ever call a variable DO20I, and that is correct. But suppose the programmer *intended* to type the DO-statement above but accidently typed a period instead of a comma (they are next to each other on the keyboard). The statement will have been transformed into an assignment to DO201. The programmer will probably not notice the error because ',' and '.' look so much alike. In fact, there will be no clue that an error has been made because, conveniently, the variable DO20I will be automatically declared. According to an oft-repeated story, an American Mariner I Venus probe was lost because of precisely this error.[9]

You should also notice that the real seriousness of this problem results from the *interaction* of two language features. If it weren't for FORTRAN's implicit declaration convention, the mistake would have been diagnosed as a missing declaration of DO20I. This is an example of a violation of the Principle of Defense in Depth, which states that if an error can get through the first line of defense without detection, then it should be caught by the next line of defense, and so forth. FORTRAN throws out what in most languages is the most significant lexical feature: the breaks between the words. Modern programming languages have lexical conventions very much like natural languages: Blanks can be (and in some cases must be) used to separate the *tokens* (words and symbols). This is both more secure and more readable.

▨ ***Exercise 2-35:*** Define a new set of lexical conventions for FORTRAN. They should permit blanks to be inserted for readability but avoid the problems we have discussed above.

▨ ***Exercise 2-36:*** Suggest a new syntax for the DO-loop that is less prone to the mistake illustrated above.

[9] Like most legends, this one is part fact, part fiction. The truth seems to be that the spacecraft was destroyed when it began to behave erratically, which was caused by a missing hyphen from an assembly language program. (The story can be found in *The New York Times* for July 28, 1962 and in *Far Travelers—The Exploring Machines*, by Oran W. Nicks, NASA, 1985.) The historical details do not alter, of course, the point of the example.

The Lack of Reserved Words Is a Mistake

FORTRAN granted programmers the dubious privilege of using as variables words with meaning in FORTRAN. For example, FORTRAN permits a programmer to have an array called `IF`:

```
DIMENSION IF(100)
```

Uses of this array

```
IF (I - 1) = 1 2 3
```

are likely to be confused with `IF`-statements:

```
IF (I - 1) 1, 2, 3
```

Again, a programmer is not likely willfully to write such a deceptive statement. The point is that when a compiler has seen '`IF (I - 1)`', it still does not know whether it is processing an assignment statement or an IF-statement. The combination of ignoring all blanks and allowing keywords to be used as variables makes the syntactic analysis of FORTRAN programs a nightmare. Consider what is required to classify something as a DO-statement. The instruction may begin with the letters 'DO', a sequence of digits, a legal variable name, and an '=' sign and still not be a DO-statement; the assignment to `DO20I` is an example. Furthermore, it is even enough to see if there is a comma to the right of the equals sign since

```
DO 20 I = A(I,J)
```

is not a DO-statement.

■ ***Exercise 2-37:*** Classify the statement on line 5:

```
      DIMENSION FORMAT(100)
  5   FORMAT(11H) = 10*(I-J)
```

■ ***Exercise 2-38*:*** Describe a procedure for distinguishing between a DO-statement and an assignment statement.

Algebraic Notation Was an Important Contribution

We will not turn from lexics to syntax. Recall that one of the goals of FORTRAN was to permit programmers to use a conventional algebraic notation. This goal was partially met; for example, the expression

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

would have to be written

```
(-B + SQRT (B**2 - 4*A*C))/(2*A)
```

This is probably about the best that could be expected considering the commonly available input-output equipment at the time FORTRAN was developed (card punches and teletypes). It is interesting to note that the Laning and Zierler system, which the FORTRAN designers had seen, provided a much more natural, interactive programming notation than FORTRAN or most other programming languages, even to this day. In any case, the provision of a quasi-algebraic notation was without doubt one of the major selling points of FORTRAN.

Arithmetic Operators Have Precedence

An important idea is the *precedence*, or priority, of an operator. This was developed so that quasi-algebraic notations would be unambiguous and have the expected meaning. For example, in mathematical notation the arithmetic expression '$b^2 - 4ac$' is equivalent to '$(b^2) - (4ac)$', that is, exponentiation and multiplication are done before addition. Also, ab^2 means $a(b^2)$, so exponentiation is done before multiplication. Considerations such as these have led to the following precedences among arithmetic operators:

1. Exponentiation
2. Multiplication and division
3. Addition and subtraction

This means that in the absence of parentheses, exponentiation is done before multiplication and division, and multiplication and division are done before addition and subtraction. Operators of the same precedence (e.g., addition and subtraction) are done in order from left to right, i.e., they associate to the left:

$$a - b + c - d = ((a - b) + c) - d$$

Languages differ on the precedence of unary operators (e.g., $-b$ and $+b$); some give them a higher priority than exponentiation, others the same priority as addition. Neither is entirely consistent with mathematical convention.

■ ***Exercise 2-39:*** Fill in the parentheses in the following expressions:

```
A + B * C - D / E
A - B - C - D
A/B/C
A/B*C
A ** B * C
I + 1 / J - 1
```

■ ***Exercise 2-40:*** Compare the precedence conventions of at least three programming languages for which you can find descriptions.

A Linear Syntactic Organization Is Used

We saw in Chapter 1 that pseudo-codes were patterned after machine languages; the instructions were listed in order in exactly the same way they were stored in memory. Numeric statement labels were used that were reminiscent of the addresses of machine instructions. For the most part, FORTRAN follows this same pattern. Statements are strung together, one after another, just like the instructions in memory; they are addressed with numeric labels. This is what we mean when we say that FORTRAN has a *linear* syntactic organization; the statements are arranged in a simple sequence ('linear' = line). You are no doubt familiar with *structured* programming languages in which some statements can be nested within others, but this was a later development. The only nesting that occurs in most FORTRAN dialects is in the arithmetic expressions and in the `DO`-loop, although the FORTRAN 77 Standard also incorporated nested `IF`-statements. We will see in later chapters that *hierarchical structure* and *nesting* are important methods for conquering the complexity of large structures.

2.7 EVALUATION AND EPILOG

FORTRAN Evolved into PL/I

In this chapter we concentrated mainly on FORTRAN IV, the dialect designed in 1962 and most characteristic of first-generation languages. This should not be taken to mean that work on FORTRAN ceased in 1962; rather, it has continued to the present. For this reason we will pick up the history of FORTRAN just after the design of FORTRAN IV in 1962.

The success of FORTRAN led to its use in many applications that were not strictly scientific. This resulted in the recognition of a serious deficiency in FORTRAN—its lack of any character manipulation facilities—and to a short-lived project within IBM to design a FORTRAN V with these facilities. Later (1963) a committee was set up by SHARE (the IBM users' group) to study extending FORTRAN so that it would be useful for both commercial and scientific applications. Although the language designed by this committee was originally known as FORTRAN VI, it soon became clear that it would be impossible to satisfy all of the goals and maintain compatibility with FORTRAN. In a preliminary specification of the language released in 1964, the language was called NPL (New Programming Language), although its name was changed to PL/I in 1965 because of protests from the National Physics Laboratory in England.

As may be expected of a language that tries to be a tool for all applications, PL/I is a very large language. The number of features in PL/I and the intricacy of some of their interactions have drawn much criticism. For example, Dijkstra has said,[10] "If FORTRAN has been called an infantile disorder, then PL/I must be classified as a fatal disease." You may be surprised to hear such virulent remarks made about a programming language, although they can be understood in the context of the improvements in programming methodology taking place at that time. Specifically, the late 1960s and early 1970s saw the development of *structured programming*, a body of programming methods intended to foster easier and more reliable programming. A complex, unpredictable, large programming language was seen as more of a hindrance than a help. For this, among other reasons, PL/I's popularity has waned in recent years, although it remains in use.

[10] *A Short Introduction to the Art of Programming.*

FORTRAN Continues to Evolve

Although structured programming is discussed in more detail in later chapters, we must understand a few things about it to appreciate the later development of FORTRAN. We have seen that FORTRAN depended heavily on the GOTO-statement as a control-structuring mechanism. Unfortunately, the undisciplined use of GOTOs can lead to "rat's nest" control structures that are hard to understand, debug, and maintain. For this reason language designers proposed higher-level control structures that obey the Structure Principle, such as the **if-then-else** and **while-do.** Although newer programming languages included these facilities, FORTRAN remained the most widely used language. For this reason many *preprocessors* (such as RATFOR) were designed that accepted these structured control structures and translated them into legal FORTRAN. These preprocessors were quite popular because they allowed programmers to take advantage of the wide availability of FORTRAN without giving up the use of the new control structures. Ultimately this led to a new FORTRAN dialect, called FORTRAN 77, which became an American National Standard. This language included the new structured control structures along with some rudimentary character manipulation facilities. The FORTRAN committee of ANSI (the American National Standards Institute) started work almost immediately on a successor to FORTRAN 77, but it took 12 years complete and was known as FORTRAN 82, 88, and finally 90 before its approval in 1991.[11] Once again the FORTRAN designers borrowed ideas that had been successful in more modern languages and adapted them to the FORTRAN framework. These included free format lexical conventions (although fixed format programs are still permitted), user-defined data types, modularization mechanisms, and pointers. (These concepts will be discussed in the following chapters, in their proper historical order.) FORTRAN 95 is a relatively minor revision of FORTRAN 90; it includes support for exception handling, parameterized types, and object-oriented programming. FORTRAN 2000 is already on the horizon. Thus, FORTRAN will continue to evolve into the new millenium.

FORTRAN Has Been Very Successful

By any standard, FORTRAN has been very successful. It accomplished its goal of demonstrating that a high-level language could be efficient enough to be used in production programs. FORTRAN is still the most heavily optimized programming language.

FORTRAN has been used effectively in almost every application area; it has shown itself to be quite amenable to extension and alteration. The awkwardness of its use in certain areas, particularly commercial data processing, led to attempts to incorporate facilities for these areas into FORTRAN; PL/I is an example of this. The lack of more elaborate data structuring methods, such as COBOL's records, and recursive subprograms has prevented FORTRAN's effective application to nonnumerical problems. For its time, FORTRAN was a triumph of design; it still holds many lessons for us.

[11] An amusing account of the development of the FORTRAN 90 standard is Brian Meek's article, "The Fortran (Not the Foresight) Saga: The Light and the Dark." *Fortran Forum 9*, 2 (October 1990), pp. 23–32.

Characteristics of First-Generation Programming Languages

In this chapter we have used FORTRAN (specifically, FORTRAN IV) to illustrate the characteristics of *first-generation* programming languages. We now pause to summarize those characteristics.

In general, we have seen that the structures of first-generation languages are based on the structures of the computers of the early 1960s. This is natural, since the only experience people had in programming was in programming these machines.

This machine orientation is especially apparent in first-generation *control structures*. For example, FORTRAN's control structures correspond almost one for one with the branch instructions of the IBM 704. First-generation conditional instructions are nonnested (unlike those in later languages), and first-generation languages depend heavily on the GOTO for building any but the simplest control structures. One exception to this rule is the definite iteration statement (e.g., FORTRAN's DO-loop), which is hierarchical in first-generation languages. Recursive procedures are not permitted in most first-generation languages (BASIC is an exception), and there is generally only one parameter passing mode (typically, pass by reference).

The machine orientation of first-generation languages can also be seen in the types of *data structures* provided, which are patterned after the layout of memory on the computers available around 1960. Thus, the data structure *primitives* found in first-generation languages are fixed and floating-point numbers of various precisions, characters, and logical values—just the kinds of values manipulated by the instructions on these computers. The data structure *constructors* are arrays and, in business-oriented languages, records, which are the ways storage was commonly organized. As with control structures, first-generation languages provide little facility for hierarchical data organization (an exception is COBOL's record structure). That is, data structures cannot be nested. Finally, many first-generation languages are characterized by a relatively weak type system; that is, it is easy to subvert the type system or do representation-dependent programming. (Machine independence and portability were not major concerns in the first generation.)

Hierarchical structure is also absent from first-generation *name structures*, with disjoint scopes being the rule. Furthermore, variable names are bound directly and statically to memory locations since there is no dynamic memory management.

Finally, the *syntactic structures* of first-generation languages are characterized by a card-oriented, linear arrangement of statements patterned after assembly languages. Further, most of these languages had numeric statement labels that are suggestive of machine addresses. First-generation languages go significantly beyond assembly languages, however, in their provision of algebraic notation. Their usual lexical conventions are to ignore blanks and to recognize keywords in context.

In summary, the salient characteristics of the first generation are machine orientation and linear structures. We will see in the next chapter that the second generation makes important moves in the directions of application orientation and hierarchical structure.

EXERCISES*

1. Study the latest ANS FORTRAN Standard and compare it with the dialect accepted by a particular compiler. Document the ways in which the implemented language differs from the standard.

2. Critique a FORTRAN Standard (FORTRAN 77 or later) with respect to the language design principles we have discussed.

3. Compare FORTRAN 77 with one or more structured FORTRAN processors (e.g., RATFOR, WATFIV).

4. Study the FORTRAN `FORMAT`-statement. Critique the `FORMAT`-statement sublanguage of FORTRAN with respect to regularity, orthogonality, structure, etc.

5. Suppose that eight decimal-digits (4 bits each) will fit in one computer word. Design a decimal arithmetic facility for FORTRAN. This must include a `DECIMAL` data type and appropriate arithmetic operations and relations. Discuss the dominance rules and coercions that will be provided.

6. Study the COBOL programming language and discuss its similarities to and differences from FORTRAN. Discuss in terms of the properties characterizing first-generation languages.

7. Some languages (e.g., BASIC) allow matrix arithmetic. Design a matrix arithmetic facility for FORTRAN, or present detailed arguments why such a facility should not be provided.

8. Find descriptions of FORTRAN II, FORTRAN IV, FORTRAN 77, FORTRAN 90, and FORTRAN 95. Make a chart showing the statements and constructs in each of these. This chart should simplify seeing the evolution of FORTRAN through the features that have been added and deleted.

9. Write to the American National Standards Institute (Committee X3J3), or look on the Internet, and find out what its current FORTRAN standardization efforts are. Critique the latest proposed standard.

10. Evaluate FORTRAN 90's support for free format programs.

11. Defend or attack the following statement: Instead of developing new FORTRAN standards that attempt to incorporate new ideas in an obsolete framework, we should freeze FORTRAN in its current state and concentrate on standardizing newer languages.

12. It has been said that FORTRAN is like the cockroach: It has survived essentially unchanged from prehistoric times, and nobody is sure why. Write an essay discussing the longevity of FORTRAN.

13. In the 1960s someone remarked, "I don't know what we will be programming in, in the year 2000, but I know what it will be called: FORTRAN." Do you think this is an accurate prediction? Discuss.