

Chapter 11

Global Control

Overview

This chapter deals with several kinds of nonlocal control structures that have come into common use. They range in sophistication from the ubiquitous `GOTO` to continuations, a form of control transfer based on higher-order functions. Between these extremes are some very useful and practical constructs: `BREAK`, labeled loops, and exception handlers.

The `GOTO` instruction is supported in many languages but completely absent in others. There are three faults inherent in the `GOTO`: bad effects on translation, bad effects on proofs of correctness, and bad human engineering properties. In spite of the fact that `GOTO` is out of favor, most procedural languages provide a `GOTO` instruction because sometimes it is the most efficient and practical way to implement a control pattern.

In the late 1960s, there was considerable controversy about using `GOTO` within a program. Dijkstra advocated that programs should be limited to a set of three basic control structures: sequence, conditional, and while loop. Each is a one-in/one-out structure making it possible to debug a program by debugging a series of shorter sections of code.

Non-local transfers of control can be important when a programmer needs to abort execution of a loop or recover from an error. In some languages, this can only be done by using a `GOTO`. In more modern procedural languages, the `break` instruction and exception handlers have been introduced to meet this need. In functional languages, non-local transfers of control can be accomplished by using continuations.

11.1 The GOTO Problem

GOTO is ancient, simple, and efficient; it mirrors the primitive jump instruction built into all machine hardware. To understand why we need modern forms of control transfer, the student must first understand what is wrong with the **GOTO**.

When the **GOTO** was first introduced into higher-level languages, it was thought to be both basic and necessary. It was considered basic because it directly reflected the branch instructions of all computers. It was considered necessary because it was used to compensate for the nonexistence of an important semantic mechanism that was not well understood until the late 1960s. Early languages such as assembly language, APL, FORTRAN, FORTRAN II, COBOL, and BASIC did not embody the abstraction “scope”. Labeled statements and/or the **GOTO** were used in place of begin-scope and end-scope markers. APL, FORTRAN II, and BASIC provided no other way to delimit the required scopes. COBOL was different. It had a more complex system with a murky kind of semantics that caused or permitted a tremendous variety of subtle errors.

Since then, there has been considerable argument about the wisdom of using **GOTO** in a program, or even including **GOTO** in a programming language. More foolish assertions have been made about this issue than about almost any other programming language feature. Use of any kind of **GOTO** instruction does cause some problems. We need to examine how this can lead to trouble, why some programmers want to use such a flawed control structure, and whether the need could be eliminated by better language design.

11.1.1 Faults Inherent in GOTO

We can divide the faults of the **GOTO** control structure roughly into three categories: bad effects on translation, bad effects on proofs of correctness, and bad human engineering properties.

Bad Effects on the Translation Process. **GOTOs** are harder to translate than structured control statements. Backward and forward branches require different techniques. A backward branch is used to form loops. To translate it, the location of each label in the object code of a program must be remembered until the end of the code has been reached, since any instruction at any distance could branch back to it. Thus the global symbol table becomes cluttered if many labels are used.

Forward branches are used to implement **IF** control structures. One conditional branch is needed at the location of the test, and an unconditional branch is needed to skip around the **ELSE** clause, if it exists. A label is needed just after the last line in the scope of a conditional. These statements are true whether applied to the code a programmer would write in **BASIC** or the code produced by a translator for a **Pascal IF**. But a forward branch cannot be translated when it is first encountered, since the location of the target label in the object code is not known at that time.

In the case of a forward **GOTO**, the location of the forward branch itself must be remembered, and later, when a label is found, the list of incomplete forward branches must be searched and any branch that refers to that label must be patched up and deleted from the list. Again, in a program with a lot of labels, this is slow and inefficient. Translation of the forward branches in an **IF** is

easier and much more efficient. The locations of the forward branches can be kept on a stack. Each **ELSE** and **ENDIF** can patch up the branch on the top of the stack. (An **ELSE** also places an item *on* the stack, to be patched by the **ENDIF**.) No searching is necessary, and since exactly one end-scope exists for each begin-scope, the stack can be popped when the top item is used.

Correctness is Hard to Prove with GOTOs. The bad mathematical properties of **GOTO** are all related to modularity. Correctness proofs start by breaking the program into one-in/one-out sections and proving that whenever some correctness property is true at the beginning of a section, it is also true at the end. Such proofs become much easier when the size of each section is small. But a program cannot be divided between a **GOTO** and its target label. Thus the more that **GOTOs** are used, and the more tangled they become, the harder it is to prove that the program does its intended job.

Some “pragmatists” are not deterred by this reasoning. They believe that mathematical proofs of correctness are seldom, if ever, useful, and that proofs are not used in “real” situations. Unfortunately, the same properties that make a program a bad candidate for a mathematical proof make it hard for a human to understand and debug. Proof and debugging have similar goals and face similar problems.

Some of the **GOTOs** might be data-dependent conditional branches. With these, the flow of control cannot be predicted by looking at the program, and the number of possible control paths that must be checked is doubled by each **IF**. With many such branches we get a combinatorial explosion of possible control paths, and complete debugging of a long program becomes highly unlikely.

The best policy for a programmer who must use **GOTO** because the chosen language lacks an essential control structure is to use **GOTO** only to emulate properly nested, one-in/one-out control structures. All forward and backward branches are kept short. This takes self-discipline and does not make translation any easier, but it avoids the debugging problems.

Bad Human Factors Inherent in the Best Use of GOTOs. When scopes are defined by labels and **GOTO** there is often poor visual correspondence between the true structure of the program and its apparent structure, represented by the order of lines on the page. **FORTRAN II** and **BASIC** had only an **IF...GO**, not a full **IF...THEN...ELSE** structure. The latter can be emulated using **GOTO**, but this has the effect of putting the **ELSE** clause under the **IF** (where control goes when the condition is false) and putting the **THEN** clause at some remote program location, often the end. The end of such a program tends to be a series of stray clauses from a bunch of unrelated **IF**'s. Things that belong together are not placed together in the source code, and the code has poor lexical coherence [Exhibit 10.9, the **IF** in **BASIC**].

Writing such code is error prone because the programmer will often write an **IF** and simply forget to write the code for the remote section. Debugging such code is also more difficult because it requires lots of paging back and forth. Patching or modifying code after it is written usually introduces even more remote sections. Eventually the flow of control can become simply too

confusing to follow.

Why Spaghetti is Unpalatable. The most difficult situation for either proof or debugging is one in which the scopes of different IFs or loops overlap partially, in a non-well-nested manner, and unconditional GOTOs thread in and out through these merged scopes. In this situation there may be a very large number of ways that control could reach a particular statement. This kind of program is called *spaghetti code* because of its tangled nature. Spaghetti code is easy to write but tricky to debug. The outstanding characteristic of spaghetti code is that virtually everything has global effects.

If one simply sits down and writes out lines of code as they occur to the conscious mind, adding afterthoughts, corrections, and improvements as needed, the result will probably be spaghetti code, even if a “structured language” is used. The lack of clarity is caused by too many IF statements asking redundant questions about obscurely named global variables. Disorganized thinking produces disorganized code, and disorganized code is hard to understand and hard to debug.

The bugs in most programs can be eliminated by cleaning up the code. The truth of this has been demonstrated over and over by beginning students, coming to a professor with programs whose bugs they cannot locate, let alone fix. The professor can look at such a program, fail to see the error right away, and suggest that a few messy parts be cleaned up and rewritten, keeping all routines short and all variables local. Having no other recourse, the student follows these orders. The student comes back an hour later with a working program. The bug never was found, but it disappeared when good “hygiene” was applied to the program. The irony is that the programmer accomplished in an hour of clean-up what had not been achieved in a whole evening of debugging through patches and modifications.

The performance of many spaghetti programs can be improved by cleaning them up. For small jobs and quick answers, the stream-of-consciousness programming method is usable and may even be fastest in the long run. For large or production jobs, it is disastrous. The resulting program is usually “loose”: it is longer, more complex, and less efficient than a well-thought-out, well-structured program.

A spaghetti program has a short useful lifetime and poor portability. In a spaghetti program with ill-defined scopes and global variables, it is difficult to define or understand how the parts of the program interact. It is, therefore, difficult to modify or extend the program in even minor ways without introducing errors into parts that formerly worked.

It is for these reasons that the business and defense communities have given their full backing to structured system design and structured programming techniques. Any program is expensive to develop, and one with a short lifetime is too expensive. Most managers do not understand programming, and they are at the mercy of their staff unless they can establish some sort of criteria for the “right way” to do things. To this end, they often require or forbid the use of certain language features, require a certain style of programming, and require extensive documentation of every step of the program development process.

Rigid rules about methodology and style often do stand in the way of modernization. New

Exhibit 11.1. Some languages with no GOTO and no labels.

Icon: Has ordinary WHILE and UNTIL loops and an IF...THEN...ELSE control structure.

Prolog: Uses the REW cycle and nested functions like LISP.

FORTH: Has structured loops and conditionals only. An assembler is an integral part of a FORTH system, but even the FORTH assembly language does not provide an arbitrary GOTO; it has only limited-distance forward and backward jumps.

Turing: Has a structured conditional and structured loops, with break.

and better languages and methods are not accepted until a new generation of managers takes over. But firm application of these rules protects a manager against unscrupulous programmers like one professional who bragged to a class that he had “tenure” at his company because his program code was both crucial to operations and undecipherable by anyone else. Perhaps to defend against programmers like this, one local company has a program that turns spaghetti code written in COBOL into modern structured COBOL code.

11.1.2 To GOTO or Not to GOTO

Against the GOTO. On the one hand it is amply clear that excessive use of GOTO and highly unrestricted GOTO instructions make programs error prone, hard to read, hard to debug, and perhaps impossible to prove “correct” by mathematical methods. The simple existence of a label in a program slows down translation. For these reasons, the computing community has largely adopted Dijkstra’s position that the use of GOTO is bad, and that structured programming, *without GOTO*, is better.

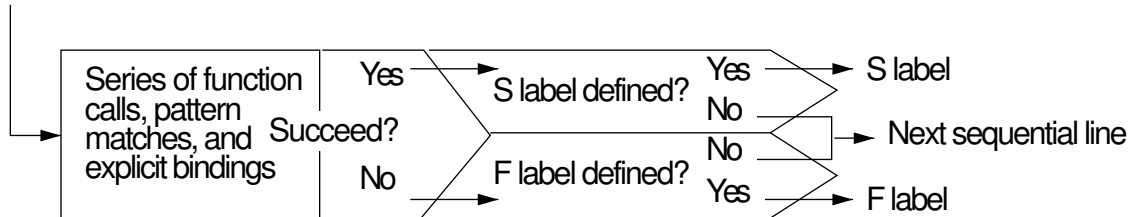
Many of the newer computer languages contain no GOTO at all [Exhibit 11.1]. The omission of GOTO is particularly interesting in the cases where a new language is basically a revision of an old one, as Icon is a revision of SNOBOL. Both are general languages, designed by R. Griswold, with particular adaptations and many functions included to make string processing easy.

SNOBOL, designed in the early 1960s, is built around an unusual one-in/three-out control structure. Each line of a SNOBOL-4 program invokes a series of function calls, name bindings, and pattern matches. Depending on the success or failure of this series of actions, processing can continue at any of three lines [Exhibit 11.2]. Even a short program can become a rat’s nest of intertwining GOTOs.

Icon is the modern successor to SNOBOL-4. The string-processing orientation remains, but the curious GOTO structure is gone. It is replaced by structured control statements, including WHILE,

Exhibit 11.2. The unique control structure in SNOBOL-4.

SNOBOL-4 was built on a unique one-in/three-out control pattern. Control flowed in at the left side of a line. Function calls, pattern matches, and explicit bindings were executed in left-to-right order. At the right end of the line, the process produced either a “success” or “failure” result, which determined the next line to be executed:



UNTIL, and IF...THEN...ELSE. Icon programs look more like C than like SNOBOL.

In Favor of the GOTO. In spite of the fact that GOTO is currently out of favor, most procedural languages do provide a GOTO instruction. This is because using a GOTO is sometimes the only efficient or practical way to implement an essential control pattern. Articles appear regularly in the popular computing magazines hotly debating the merits of the GOTO instruction and exhibiting programs that cannot be written efficiently (in languages such as Pascal) without it.

It is important to understand that if the control structures of a language reflect the commonly occurring process control patterns, then the programmer will be happy to use them instead of GOTO. Some languages are clearly deficient in control structures. For example, FORTRAN lacks a WHILE loop and a CASE statement, and BASIC lacks a WHILE loop, an IF...THEN...ELSE conditional, and procedure parameters. Textbooks for beginning programmers in these languages have been rewritten to preach the use of structured programming, even though adequate one-in/one-out control structures are not provided by the language.

Authors of textbooks for both languages advocate achieving “structure” by restricting the use of labels and GOTO to emulate the standard IF and WHILE control structures in Exhibit 10.2. Unfortunately, this produces FORTRAN code that is awkward, cluttered, and unmotivated by any real need. Dijkstra’s control structures are not the only possible “good” ones and are not the most appropriate to superimpose upon a GOTO language.

Pascal contains all of Dijkstra’s control structures, but these form a very limited tool bag for the expert programmer. One might suggest that limiting the programmer to WHILE, FOR, REPEAT, IF...THEN...ELSE, and CASE is like limiting a skilled carpenter to the use of a screwdriver, hammer, wrench, saw, and a broken chisel. Perhaps the programmer can do the job required, but not in a very pretty way. These five control structures satisfy programming needs much of the time, but there are many common situations in which they fall short. Pascal programmers who use GOTO do so primarily when they wish to implement the following four control structures efficiently:

- An OTHERWISE clause in the case statement.
- A loop statement whose exit test can be placed anywhere in the scope [Exhibit 11.4].
- A break statement [Exhibit 11.9] to be used in loops, or a two-exit loop, returning a “success” or “abort” result.
- A way to terminate a nest of loops all at once after an error [Exhibit 11.10, Ada loop labels].

As long as programmers work with languages such as FORTRAN and Pascal, GOTO will be used to emulate important missing control structures, and we will continue to hear arguments from naive programmers about the value and indispensability of GOTO. In a properly designed language, GOTO has few, if any, uses.

Ada has the control primitives, missing in Pascal, to handle the needs described. Consequently, there is little demand for the GOTO in Ada. It is not possible to give a list of all the control structures that might sometime be useful, but we can provide enough structures to satisfy 99.9% of the needs, and make 99.9% of the uses of GOTO unnecessary. It remains a design issue whether a language should provide GOTO to allow the programmer to write efficient code for that remaining .1% of situations.

11.1.3 Statement Labels

A discussion of GOTO must cover the target of the GOTO—a statement label. Programming languages have used a variety of methods to define labels, including label declarations, simple use, and merging labels with source code line numbers.

Line Numbers Used as Labels. Some interactive languages, notably BASIC and APL, took a shortcut approach to statement labels: the same line numbers that define the sequence of statements were used as the targets for GOTO. Like all shortcuts, this caused real semantic trouble.

First, it deprives the programmer of any way to distinguish lines that form the top of a program scope from lines that are simply in the middle of some sequence of commands.¹ A programmer normally knows which lines of code should and should not become the target of a GOTO. Even a programmer who chooses to create spaghetti code with tangled scopes knows where each thread should and should not begin. A language that provides no way to communicate this information is not offering the programmer a good vehicle for modeling a problem.

Second, as discussed in Chapter 2, Section 2.3.3, line number/labels are a compromise because the semantics needed for line numbers are different from those needed for labels. If the semantics defined for them are appropriate for use by the on-line editor, then the semantics are inappropriate for defining flow-of-control, and vice versa.

¹Discussed more fully in Chapter 2, Section 2.3.3.

Exhibit 11.3. The form and definition of labels.

Language	Form	Rules for making and using labels
Ada	alphanumeric	Place << label_name >> before any statement. Use in the GOTO statement.
APL	alphanumeric	Place label followed by “.” before any statement. Use in “→” statement.
COBOL	alphanumeric	Place beginning of label in columns 8–11 (ordinary statements cannot start before column 12). Use in the GOTO or PERFORM statements.
FORTRAN	numeric	Place in columns 1–5 at the beginning of any executable statement. Use in the GOTO statement.
LISP	alphanumeric	Labels and “go” expressions are legal only within a “prog” or a “do” body. The label is defined by an expression that forms part of the body of the prog. The expression “(go label_name)” within a prog body causes transfer of control to the expression after label_name.
Pascal	numeric	Declare label at top of program and place it, followed by “.”, before any statement. Use in GOTO statement.
PL/1	alphanumeric	Place label followed by “.” before any statement. Use in GOTO statement.
SNOBOL-4	alphanumeric	Place label and “:” at the left side of any statement. Use labels by placing their names in the “succeed” or “fail” field of any statement.

Defined Labels. Most languages avoid these problems by having a programmer simply write a label at the beginning of any line. In **Pascal**, the label must also be declared at the top of the program. (This permits **Pascal** to be compiled easily in one pass through the source code. Languages without declared labels either require two passes or some complicated patching-up of the generated code at the end of the first pass.) Exhibit 11.3 summarizes the rules for defining and using labels in several common languages.

Conclusions. The uses and misuses of **GOTO** and statement labels have been considered, along with a variety of ways to minimize the use of **GOTO**. These include the obvious ways—inclusion of structured loops and conditionals in the language, and ways that have only recently been appreciated, including providing more flexible loop control structures and a structured **BREAK** statement which replaces the unstructured **GOTO**.

Exhibit 11.4. An input loop with an exit in the middle.

A simple interactive read loop is shown here in FORTRAN, demonstrating a common situation in which the loop exit is placed at a position other than the top or bottom:

```
100 PRINT ('Please type your name.')
    READ *, (NAME)
    IF (NAME .EQ. 'Quit') 200
    PRINT ('Hello ', Name, 'Nice to see you.')
    GOTO 100
200 PRINT ('Goodbye.')
```

Compare this code to the Pascal version in Exhibit 11.5.

11.2 Breaking Out

Many loops have two possible reasons for termination, not one. The basic search loop is a good example: control will leave the loop after all data items have been searched and the key item is not found anywhere. This is called a *normal exit*, or FINISH. On the other hand, if the item is in the data base, a search loop should terminate as soon as it is found, before examining all the remaining items. This is called an *abortive exit*, or ABORT. In a search loop, a normal FINISH is associated with failure and an ABORT with success.

As another example, an input loop will ABORT if a premature end-of-file is encountered and will FINISH if the number of input items meets expectations. In this case, the FINISH is associated with success and the ABORT with failure.

A loop syntax that provides two ways to leave the loop is important. It is applicable to the large number of situations in which a repeated process can either FINISH or ABORT. To write such a loop using GOTO is easy. The normal termination condition can be placed in the loop control element; the ABORT condition is placed in an IF-GO within the loop [Exhibit 11.4].

To accomplish the same thing without using IF-GO is more complex. In the case of an input loop, this is generally handled by using a priming read [Exhibit 11.5]. The prompt and input statements are written twice, once just before the loop, guaranteeing that the loop variable will be initialized, and once at the end of the loop, to prepare for the next test. This is correct and executes efficiently, but it is clumsy to write.

An inexperienced programmer thinks that he can solve this problem by using an UNTIL loop instead of a WHILE loop. Unfortunately, this just leads to a different kind of repetition: the loop exit condition has to be tested within the loop (as well as at the end) in order to avoid processing nondata when the end-of-file condition occurs [Exhibit 11.6]. This is even less elegant than the priming read, since it introduces an extra level of nested logic and it causes unnecessary duplication of work on every iteration.

Sometimes two exit conditions can be combined into a single loop control expression using AND.

Exhibit 11.5. The Pascal input loop with a redundant priming read.

```

Writeln ('Please type your name. ');
Readln (Name);
While Name <> 'Quit' do begin
    Writeln ('Hello ', Name, 'Nice to see you. ');
    Writeln ('Please type your name. ');
    Readln (Name);
end;
Writeln ('Goodbye. ');

```

Compare this code to the FORTRAN version in Exhibit 11.4.

In other cases this is not possible, either because some action must be done between testing the conditions, or because the second condition would cause a run-time error whenever the first is true [Exhibit 11.7].

The WHILE loop in Exhibit 11.7 translates into the following steps:

1. ($k \leq \text{max}$) is evaluated (call the result $t1$).
2. ($a[k] > 0$) is evaluated (call the result $t2$).
3. ($t1$ and $t2$) is evaluated (call the result $t3$).
4. If $t3$ is false the loop exits, otherwise $\text{sum} = \text{sum} + a[k]$ and $k := k + 1$ are executed.
5. Control returns to step 1.

This code segment will “bomb” with a “subscript out of range” error at the end of summing any array whose elements are all positive. What happens is this: when (1) is false, (2) is evaluated anyway, and since k is too large, this causes an error. (In fact, that is why test (1) was included.)

To program such a loop correctly in Pascal one must put condition (1) in the loop control element and condition (2) in a separate statement inside the loop. A superfluous Boolean variable

Exhibit 11.6. The Pascal input loop with an extra test.

```

Repeat
    Writeln ('Please type your name. ');
    Readln (Name);
    If Name <> 'Quit' Then Writeln ('Hello ', Name, 'Nice to see you. ');
Until Name = 'Quit';
Writeln ('Goodbye. ');

```

Exhibit 11.7. Some Pascal code that will stop with a run-time error.

Assume *k* is an integer, and *a* is an array with subscripts from 1 to *max*. This code sums the items in *a* up to but not including the first negative or zero element.

```
k := 1; sum := 0;
While ( k<=max ) and ( a[k]>0 ) Do Begin
    sum := sum + a[k];
    k := k + 1
End;
```

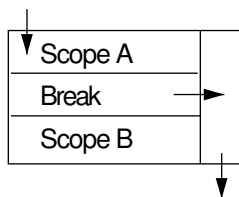
is declared, and an IF statement within the loop sets it to TRUE if the ABORT condition happens. Then the test in the loop control element is modified to test this flag also. This will cause an exit from the loop at the beginning of the iteration after the ABORT condition was discovered. After loop exit, some other mechanism must be found to determine what condition caused the exit and take appropriate action. This awkward program idiom is illustrated in Chapter 8, Exhibit 8.13, where a flag named *done* is introduced to permit an abortive loop exit.

In more complex situations a Pascal programmer might use several Boolean flags, set in several places, and combined into a complex expression for the exit condition. This is logical spaghetti just as surely as if we used many GOTOs to create this control pattern. Our goal is to eliminate the need for spaghetti in our programs.² To do this we need a structured way to exit from the middle of a loop. Such an exit mechanism is provided by the BREAK statement (or its equivalent) in many languages. The BREAK is simply a structured GOTO to the end of the control unit. It may be placed anywhere within a loop, and it transfers control immediately to the statement following the loop. Its control diagram [Exhibit 11.8] is an immediate exit into the frame of the statement. (A frame is added if the control structure does not already have one.)

²See Section 11.1.

Exhibit 11.8. Structured conditional loops with BREAK.

BREAK exits to the right.



A loop with a conditional BREAK.

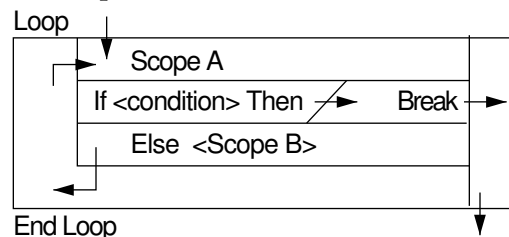


Exhibit 11.9. The input loop implemented using break in C.

```

char *name;                /* This is a string variable. */
for(;;)                    /* An explicit endless loop. */
{
    printf ("Please type your name. ");
    scanf ("%s", name);
    if (name[1] == '\0') break; /* Exit if no name was entered. */
    printf ("Hello %s, Nice to see you.\n", name);
}
printf ("Goodbye.");

```

C³ and the newer procedural languages, Ada and Turing, all include some form of the **BREAK** statement for aborting a loop in a structured manner. In Ada and Turing it is called **EXIT**. This is shown on a control diagram as a frame along the right side of the box which is connected to the bottom frame, from which control leaves the entire diagram. Control enters this frame along a horizontal arrow wherever a **break** occurs in the program.

Exhibit 11.9 gives one final version of our print routine, written in C using **break**, which causes immediate exit from a loop. By putting a **break** instruction inside an **if**, and embedding the whole thing in an infinite loop, we can achieve the same thing as in the original FORTRAN example, simply and efficiently.

11.2.1 Generalizing the BREAK

In addition to statement labels, for use with the **GOTO** instruction, Ada also has loop labels, which are used to create double-exit loops and to allow immediate, one-step exit from a nest of loops. The simple **BREAK** statement takes control immediately out of the enclosing control structure, and into the control scope that surrounds it. The Ada **EXIT** is a generalization of this idea; it can cause an immediate exit from an entire nest of loops (or any part of a nest) by naming the loop label of the outermost loop. It can be used for any kind of loop (**FOR**, **WHILE**, or infinite)⁴ [Exhibit 11.10]. We will call this control structure the **FAR_EXIT**. The canonical example of an application for the **FAR_EXIT** is a search loop over a two-dimensional data structure such as a matrix or a list of lists. A double loop is used to process the structure, and when the key item is found, control must leave both loops.

Implementing this exit actually requires more than a simple **GOTO**. When an Ada **FOR** loop is entered, stack space is allocated for the loop variable. This obviously must be deallocated upon exit

³C also has a **continue** statement, which can be used inside any loop. It transfers control immediately to the top of the loop and begins the next iteration.

⁴Ada is the only major language that lets you name a loop scope so that you can exit from it.

Exhibit 11.10. Loop labels in Ada.

In the following code, a dictionary is represented as a linked list of words, and each word is represented as a linked list of characters. The double loop searches a dictionary for the first word containing the character '#'. Upon loop exit, `word_pointer` and `char_pointer` either point to the required word and the '#', or to null, if no such word was found. These variables must be defined outside the `outer_loop`.

```

outer_loop: WHILE dictionary_pointer /= null
              word_pointer := dictionary_pointer.word;
              char_pointer := word_pointer;
              WHILE char_pointer /= null
                IF char_pointer.letter = '#'
                  THEN EXIT outer_loop;
                  ELSE char_pointer := char_pointer.next
                END IF;
              END LOOP;
            END LOOP outer_loop;
            -- Control comes here from the EXIT above.

```

from the loop. A `FAR_EXIT` can be placed at any nesting depth. When executed, it will terminate all loops in the nest, up to and including the labeled loop, and any other loops that were placed within that outer loop. The obvious implementation of this action is to scan backwards through the stack, deallocating everything up to and including the storage for the labeled loop. This kind of sudden exit can be quite useful for error handling but must be accompanied by some way to pass information outward, to the calling program, about the reason for the sudden exit. Variables that are global to the outer loop are used for this purpose in Ada.

11.3 Continuations

All the nonlocal control structures discussed so far have been defined in terms of sequences of statements. Functional languages do not have such sequences but must supply some way to accomplish the same ends in a functionally acceptable manner. Programmers who use functional languages still must write search loops and make sudden error-exits. The continuation is a functional object that meets this need.

Briefly, a *continuation* is a function which “acts like” the remainder of the program which is still to be executed. For example, suppose a programmer writes a program to compute $\sqrt{2}$ and then print out the answer. We can think of this program as having two parts:

1. The job of the square root function is to compute a real number which, when multiplied with

Exhibit 11.11. Packaging a continuation in Scheme.

```
(define (makeecc)
  (let ((x (call-with-current-continuation
            (lambda (cc) (set! gcc cc) '()) )))
    (display "returned value: ")
    (display x) ))
```

itself, equals 2.

2. The print routine waits to receive a number from the square root function, at which time it prints the number it receives on the user's screen.

Step 2—the part of the program that will make use of the answer provided by part 1—is called the continuation of the program. You can think of the term “continuation” just as a fancy name for “the rest of the program still to be executed”.

The concept of a continuation exists in all programming languages and is of concern to the language implementor, but only languages that support higher-order functions are able to give the programmer explicit access to the continuation. This is because the continuation acts like a function. It waits to receive a value and then performs an action or computation which depends on that value. Manipulating a continuation is like manipulating a function. If functions are already first-class objects, then continuations can be treated like any other function. If not, then special language primitives would be required. It is not surprising that most languages elect to omit such facilities.

Even in a language that supports higher-order functions, a primitive is needed to package up the current continuation in a form that the program can use. Packaging the continuation is like establishing a *checkpoint*—everything must be saved that is needed in order to safely restart execution at a later time. This typically includes the program counter, the stack, the environment, and so forth. Saving the current continuation can be quite an expensive operation indeed!

We will use Scheme syntax to illustrate the formation, semantics, and use of continuations. In Scheme, this primitive is called *call-with-current-continuation*, which is often shortened to `call/cc`. It is a higher-order function which takes another function, f , as its single argument. It creates a one-argument function, cc , from the current continuation and then calls f with argument cc . Function f is an arbitrary program, usually written by the programmer, which may or may not make use of the function cc . If cc is ever called, the current environment is discarded and replaced with the one saved when cc was created, and the argument to cc is passed back as the value returned by `call/cc`.

In Exhibit 11.11, we show how to use `call/cc` to store the current continuation in a global variable, “gcc”. The argument given to `call/cc` is the nameless function defined by “(lambda (cc) (set! gcc cc) '())”. `call/cc` applies this lambda expression to the current continuation,

which results in the local symbol `cc` (in the body of the lambda expression) being bound to the current continuation. This lambda expression doesn't do much—it just assigns its argument to a global variable named `gcc`, which exports the continuation into the surrounding environment. Finally, the lambda expression returns a null list to `call/cc`, which passes the value on to its caller, the `let` expression, which binds the symbol `x` to the return value, in this case, the null list. The display statements cause the comment “`returned value`” and the value of `x` to be printed on the terminal.

After returning from `call/cc`, the name `gcc` can be used, any time, to call the saved continuation. If the user types in the command “`(gcc 3)`”, she or he will see the response “`returned value: 3`”. The reason this happens is that `gcc` returns control to the previous control point where `call/cc` was about to return with the null list, but this time `call/cc` returns 3 instead. Thus `x` gets bound to 3. Execution now continues as before, printing the comment and the current value of `x`.

This example just demonstrates how a continuation may be formed and called; it does not do anything very useful. However, the `call/cc` is a powerful control structure which can be used to emulate other control structures, including `WHILE` loops and `BREAK`. This ability to break out of a piece of code is the most common use for `call/cc`. If one thinks of execution over time, the continuation is created when you execute the `call/cc``CALL/CC`, *just before* `f` is called, and the continuation consists of the code that will be executed *after* `f` returns. The execution of `f` itself is not included in the continuation; thus calling the continuation from within `f` has the effect of causing control to jump ahead to the point it would be if `f` were allowed to finish normally.

In the Scheme code in Exhibit 11.12, the function named `search` implements the search procedure by making a continuation which is used to break out of the search process when the search key is found in the data list. `Search` creates a continuation, binds it to the symbol `cc`, then calls `scan` with `cc` as an argument. `Scan` performs a recursive search and uses its continuation-argument (which it calls `foundf`) to break out when the key is found. `Scan` returns the rest of the list starting with the key.

The continuation requires an argument, so when it is called by `scan`, the answer found by `scan` is passed to the continuation, which in turn passes it on to the calling environment. A transcript of a test of `search` is shown in Exhibit 11.13. You can see that calling the continuation really does abort the recursive search.

In the search example, the continuation was used to do a forward jump, skipping over work that would otherwise have been done. However, there is nothing in the definition of a continuation that restricts its use to jumping forward in time. It can also be used to return to a previously visited point of the computation. Suppose, in the above example, that `cc(w)` were called, not while executing `f` but while executing `cc` itself. This seems circular, and indeed it is! This causes `cc` to start over again, but this second execution is given argument `w`, whereas the first execution might well have been with some other argument. (If not, the program will be in an infinite loop!)

The next two examples show how `call/cc` can be used to emulate `WHILE`. Although Scheme has an iterative control structure, it is not a pure functional construct. `Call/cc` can be used instead, in order to make Scheme truly behave like a pure functional language.

Exhibit 11.12. Call/cc used in Scheme to break out of a loop.

```

; Search a list, break when found. Returns tail of list, starting
; with the atom equal to the key.
(define (search key ls)
  (call-with-current-continuation      ; label break point
    (lambda (cc) (scan key ls cc)))) ; calling cc causes break

; Perform actual search. Called like search, but "foundf" is a
; functional argument that is applied to the part of the list
; starting with the first occurrence of the search key.
(define (scan key ls foundf)
  (if (null? ls)                                ; if end of list
      '()                                       ; return failure token
      (begin                                   ; else
        (display "examining element ")        ; document progress
        (display (car ls))
        (newline)
        (if (eq? key (car ls))                ; test list head
            (foundf ls)                       ; break out if found
            (scan key (cdr ls) foundf))))))   ; else recursively scan tail

```

A transcript of a test run is shown in Exhibit 11.13

Exhibit 11.14 shows a simple counting loop, implemented using a continuation. At the top of the routine, a continuation is formed and bound to the name “cc”. The continuation marks the spot to return to later. (This is analogous to what a compiler must do to translate the beginning of a WHILE loop.) The result of `call/cc` is a list containing this continuation and a number. The code analogous to the body of the loop starts with `let`; the first thing it does is to give names to the items in the list that was returned by `call/cc`; the continuation is named `gcc` and the number is named `n`. Now, this number is tested. If it exceeds the limit, a “!” is displayed and execution ends because there is nothing more to do. If `n` is less than the limit, it is displayed and the continuation is called (with `n` incremented) to continue the looping process. Test results are shown at the bottom of the exhibit; the command `(count 5)` does, indeed, count to 5.

Exhibit 11.15 shows another function: one that computes factorial. Compare this to the Pascal version of factorial in Exhibit 11.16.

When used in this manner, the continuation becomes like a WHILE statement: it returns repeatedly to a fixed starting point, each time bringing with it a value that can be tested and can potentially stop the repetitions. Each time the continuation is called, the new context *replaces* the old context, just as the new value of the WHILE loop variable replaces the old value. Here is a wolf

Exhibit 11.13. Testing the break routine.

This is a transcript of a test run of the code from Exhibit 11.12, translated by MIT Scheme. The characters “1]=>” are Scheme’s prompt.

```

; Define some test data
1 ]=> (define boys '(tom dick harry bob ted))

1 ]=> (search 'dick boys)
examining element tom
examining element dick
;Value: (dick harry bob ted)

1 ]=> (search 'john boys)
examining element tom
examining element dick
examining element harry
examining element bob
examining element ted
;Value: ()

```

Exhibit 11.14. Using call/cc to make a loop.

```

; (count n) prints the numbers 1...n ! on a line.
(define (count limit)
  (let ((x (call-with-current-continuation
            (lambda (cc) (cons cc 1))))))
    (let ((gcc (car x))
          (n (cdr x)))
      (cond
        ((> n limit) (display "!"))
        ((begin
          (display n)
          (display " ")
          (gcc (cons gcc (1+ n))))))))))

1 ]=> (count 5)
1 2 3 4 5 !
;No value

```

Exhibit 11.15. A Pascal-like implementation of factorial, using a continuation.

```

(define (fact n)
  (let ((x (call-with-current-continuation
            (lambda (cc) (list cc 1 1))))))
    (let ((gcc (car x))
          (k   (cadr x))
          (p   (caddr x)))
      (cond
        ((> k n) p)
        ((gcc (list gcc (1+ k) (* k p)))))))
1 ]=> (fact 5)
;Value: 120

```

in sheep's clothing: destructive assignment disguised as a higher-order function in a form that is "politically correct" for a functional language.

This kind of "backing-up" behavior is appropriate in several real-life situations. The Prolog interpreter backs up whenever the tentative instantiation it has been trying fails. A transaction processing system backs up if the transaction it is in the middle of processing is forced to abort. An operating system backs up the I/O system when a disk read error occurs and tries the read again.

Exhibit 11.16. The factorial function in Pascal.

```

function fact (n: integer) : integer;
var
  temp: integer;
  i: integer;
begin
  temp := 1;
  for i := 1 to n do temp := temp * i;
  fact := temp
end;

```

11.4 Exception Processing

11.4.1 What Is an Exception?

An *exception* is a situation that makes it impossible for a program to proceed with normal processing. It is an unpredictable situation with which the program must cope. Exceptions happen frequently enough so that a robust system cannot ignore them, but seldom enough so that they are unusual. There are three major ways in which exceptions arise:

1. A hardware error occurs and triggers a hardware interrupt signal. Examples of hardware errors are highly varied and well known to all programmers. These include: an attempt to divide by zero, numeric overflow during an arithmetic operation, an attempt to access protected memory or a nonexistent address, a disk-read error, I/O device not ready, and others.
2. A system software module identifies an error situation. Examples include: a nonnumeric character entered in a numeric input field, an attempt to use a subscript outside of defined array bounds, an attempt to store an out-of-range value in a subrange-type variable, an attempt to open a file that does not exist, an attempt to write a file on a device that has no available space, an end-of-input file, and so on.
3. A user function identifies a logically impossible or inconsistent situation. Examples, of course, are specific to the application. Some might be: a request for processing on a data item that is not in the data base, illegal data value in input stream, and the like.

A program that ignores exceptions invites disaster. For example, arithmetic overflow can be ignored in COBOL, and will be, unless the programmer explicitly tests for it. The result is that the COBOL program continues its normal computations using a meaningless bit pattern that looks like data and is likely to print out a meaningless answer. Ignoring an end-of-file condition in a C program will generally result in an infinite loop processing the last data item repeatedly. The worst possible thing to do about an exception is pretend that nothing is wrong and keep processing.

A second possible response to an exception is to terminate the program immediately and return control to the system. This action is what the Pascal standard *requires* as a response to a subrange error, subscript error, or arithmetic overflow. While this may be an acceptable way to manage faulty student programs, abrupt termination is not acceptable for a production system. At the very least, relevant information about the state of the system and the cause of the error should be made available.

In the old days, a sudden termination of this sort would usually be followed by an octal or hexadecimal dump. The programmer could then manually decode and trace the action of the program and perhaps find the cause of the sudden termination. A more modern response, typical in Pascal systems, is the “call traceback”, which dumps the chain of dynamic links from the run-time stack. Unfortunately, a dump is a primitive tool for tracing errors, and a “call traceback”

offers too little information. Further, even if the programmer can decode these communications, the end user, who must handle the error, generally cannot.

Robust systems must be able to recover from errors. Ignoring exceptions and sudden death are both unacceptable alternatives. Exceptions must be identified, if possible, and their effects controlled and limited as much as possible. Often, if the exception is handled at the right level, corrective action can be taken and the program can finish normally. For example, if a user of an interactive program types an illegal input, it should be identified, and the user should have an opportunity to retype the data. Sometimes the length of the code to handle exceptions exceeds the length of the code to process normal data.

11.4.2 The Steps in Exception Handling

There are four steps in processing an exception condition:

- Detect the exception.
- Pass control to the unit that is able to take action.
- Handle the exception by taking constructive action.
- Continue with normal execution after correcting the problem.

The last two steps are straightforward and can be done with ordinary program code, so long as the language provides support for the first two steps. Unfortunately, support for steps 1 and 2 is spotty and limited in most current languages.

Detecting Hardware and Software Exceptions. Computer hardware detects many kinds of errors automatically and generates hardware interrupts to signal that an exception has happened. An interrupt is processed by the operating system which generally sets a status flag in response. These status flags can be read by application programs at run time, but many languages do not provide language primitives to do so. Thus the warning provided by the hardware and the operating system is simply ignored.

Even in languages as early as COBOL, the importance of error containment was recognized. Arithmetic commands could contain an optional `ON SIZE ERROR` clause which provided a way to send control to an error routine. After processing the error, control could return to the next step and the program could continue. PL/1 was another early language with some support for exception handling. It has defined names for system-generated hardware and software exceptions. The user can define handlers for these exceptions using an `ON CONDITION` clause. In case of an error, control would come to a user-defined routine, giving the programmer an opportunity to print appropriate error comments. For most errors, however, the programmer cannot prevent program termination after printing comments.

One of the glaring omissions in standard Pascal is any way to test hardware status codes. Arithmetic overflow, division by zero, and the like cannot be either detected or handled within the user program. Because error detection and recovery is an essential part of a production program,

most commercial Pascal implementations extend the language by introducing predicates which test the system status flags.

Some languages use error codes, returned by system subroutines, to provide the programmer with information about exceptions. This is a common way to handle I/O errors. For example, in the standard C library, input, output, file-open, and dynamic memory-allocate commands all return error codes for exceptional conditions. The user's program may ignore these codes, which is risky, or take action.

Finally, some languages provide a general exception-handling control structure, along with a list of predefined exception names. The programmer may test for these names in the program, and the control structure delivers control to user-defined exception-handling code. In Section 11.4.3 we examine this exception facility in *Ada*.

Passing Control. The user often knows what can and should be done to recover from or correct an exception error. However, a good program is written as a series of modules; the exception may be discovered in a low-level module that is called from many places. That module often cannot take constructive corrective action because it does not have access to important data objects. Intelligent processing can be done only in a context where the meaning of the data objects is known and the severity of the exception can be evaluated. For example, in an interactive program, an "invalid input" exception may be detected in a deeply nested subroutine, but only the main interactive module can interact with the user.

Knowledge of the exception must be passed back from the routine that discovered the exception to the point at which it can be handled gracefully. There are two ways that control can be transferred in a language with no special exception control mechanism: by using *GOTO* or a continuation, or by propagating an error code back up a chain of nested subroutine returns.

GOTO is not a good solution. Although it passes control, it cannot pass information about the exception with that control. Any such information has to be passed through global variables. Further, most languages have restrictions about the target of a *GOTO*; for example, control cannot pass into the middle of a loop or a conditional, and it can only go to a label that is visible in the scope of the *GOTO*. The result is that exception handlers end up being defined at the top level, so that they are accessible. After taking action, it is then difficult to resume normal operation.

In a functional language, continuations can be used to handle exceptions in much the same way as a *GOTO* can be used.

Passing an error code back up a chain of calls can be done in any language, and the return value can encode information about the exact nature of the exception. We call this process *error propagation*. Control can be passed backwards to the right level for handling the problem. One difficulty of this approach is that the error handling code must be intermixed with the code for normal processing, making the logic for both cases harder to write and comprehend. A more serious difficulty is that the error propagation code must be present in every subroutine in the chain between the detecting module and the handling module. These intermediate routines have no interest in the error; they did not cause it and don't know how to handle it, yet they have to

Exhibit 11.17. I/O with error checking in C.

```
void silly()
{   int age, code;
    char * name;
    FILE * infile, outfile;
    if ( !(infile = fopen("myinput", "r"))
        { fprintf(stderr, "Cannot open input file.\n"); exit(1);}
    if ( !(outfile = fopen("myout", "w"))
        { fprintf(stderr, "Cannot open output file.\n"); exit(1);}
    if ( fscanf(infile, "%s%d", name, &age) <2)
        fprintf(stderr, "Bad data record ignored\n");
    if (fprintf(outfile, "Hi %s; I hear you are %d years old!\n", name, age)
        == EOF)          { fprintf(stderr, "Cannot write to file myout!\n"); exit(1)
    }
}
```

The system call `exit(1)` causes the program to abort gracefully, flushing all buffers and closing all open files.

pass it on. This greatly clutters the code and obscures its logic. The code itself is very error prone.

For example, a well-written C program that uses the standard I/O library will check the code returned by every I/O statement and take corrective action if the result differs from the expected result. In Exhibits 11.17 and 11.18, two versions of an I/O routine are given—a robust version and a clear version. The error-trapping code makes the robust version much longer and more obscure; the actual I/O operations are buried in a mass of exception code. This demonstrates the importance of an exception control structure in a modern language.

Exhibit 11.18. I/O without error checking in C.

```
void silly()
{   int age, code;
    char * name;
    FILE * infile, outfile;

    infile = fopen("myinput", "r");
    outfile = fopen("myout", "w");
    fscanf(infile, "%s%d", name, &age);
    fprintf(outfile, "Hi %s, I hear you are %d years old!\n", name, age);
}
```

11.4.3 Exception Handling in Ada

The facility for exception handling in Ada provides a real contrast to the do-it-yourself versions we have discussed. Ada provides excellent support for detection of both hardware and software generated exceptions. It provides reasonable, clean solutions to the problems of passing control, handling exceptions, and restarting normal processing.

The exception facility includes four kinds of statements:

- A declaration form for user-defined exception names.
- A command to signal that an exception has happened.
- A syntactic word, “EXCEPTION”, to separate the regular program code from the exception handlers.
- A syntax for defining a procedure to handle one kind of exception.

Exhibit 11.19 shows samples of each kind of exception statement. In the following discussion, we examine the usage of each part of the exception system.

User-defined Exceptions. Descriptions of the Ada language and the standard library packages list the names and semantics of the exceptions that the Ada system might generate. In addition, programmers can define their own exceptions, which will be handled the same way as those defined by the system. A simple declaration defines a name as an exception:

```
⟨exception name⟩ : EXCEPTION;
```

To signal a user-defined exception, we use the RAISE statement:

```
RAISE ⟨exception name⟩;
```

Exception Handlers. An *exception handler* is a body of code labeled by the name of an exception. A program can have handlers for user-defined exceptions and/or for system-defined exceptions. A programmer writing code that must be reliable needs to define handlers for any of those exceptions that might be raised by that job. Any block of code (BEGIN...END) can have local exception handlers, which are declared at the bottom of the block. The syntax for defining an exception handler is:

```
BEGIN
    ⟨code for processing normal case⟩
EXCEPTION
    WHEN ⟨exception name⟩ =>
        ⟨action for that exception⟩
    WHEN ...
    WHEN OTHERS =>
END
```

Exhibit 11.19. Handling an integer overflow exception in Ada.

The simple program `Circles` accepts interactive input and calls procedure `One_Circle` to process it. `One_Circle` calls an output routine, `Print_Circle`, which is omitted here for brevity. If an arithmetic overflow occurs during execution of `One_Circle`, normal processing of that data is aborted: `Print_Circle` is not called, and the normal prompt is not printed. Instead, the user is prompted to reenter the data, and processing proceeds with the new data.

```

WITH Text_IO; USE Text_IO; PACKAGE Int_IO IS NEW Integer_IO(integer);
-- -- -- -- --
PROCEDURE Circles IS
  Diameter: Integer;
BEGIN
  Put ("Please enter an integer diameter.");
  LOOP
    BEGIN
      Int_IO.Get(Diameter);
      EXIT WHEN Diameter<=0;
      One_Circle(Diameter);
      Put ("Please enter another diameter (0 to quit): ");
    EXCEPTION
      WHEN Diameter_Too_Big_Error =>
        Put("Invalid input--diameter too large. Please reenter: ");
    END
  END LOOP;
  Put ("Processing finished.");
  Put New_line;
END Circles;
-- -- -- -- --
PROCEDURE One_Circle (Diam: Integer) IS
  Radius, Circumference, Area : Integer;
BEGIN
  Circumference := Diam * 22 / 7;
  Radius := Diam / 2;
  Area := Radius * Radius * 22 / 7;
  Print_Circle(Diameter, Radius, Circumference, Area);
EXCEPTION
  WHEN Numeric_Error =>
    RAISE Diameter_Too_Big_Error;
END One_Circle;

```

The exception portion of the block can contain as many `WHEN` clauses as needed, and their order does not matter. (These clauses must be mutually exclusive, though; one exception name cannot be used to trigger two clauses.) A program that wishes to be sure to catch all exceptions, even unexpected ones, can include a handler labeled `WHEN OTHERS =>`. In the `Circles` example, both the main program and the subprogram have one exception handler.

Passing Control and Resuming Normal Processing. The code of a handler is translated in the context of its enclosing block; that is, variable bindings and visibility are the same as for the rest of the code in the block. The block `BEGIN...END` also defines the scope of the abort action associated with each exception: all code is skipped from the point at which the exception is raised until the end of the block, and the exception handler is executed in its place. If an exception is raised during execution of a subroutine call, all statements in the block after the call will be skipped.

Raising an exception starts an abort action. If there is no local handler for the exception, the current block is aborted and its frame is removed from the run-time stack. The exception is then raised in the routine that called the aborted routine. If that routine has a handler, it is executed. Otherwise, it is aborted and the exception is passed backward another level. This aborting continues, removing one stack frame each time, until a stack frame is found that contains a handler for the exception. The handler is executed and stops the popping. If a program has no handler for an exception, all frames will be popped, the main procedure will be aborted, and control will return to the system.

Raising and Propagating Exceptions. In procedure `One_Circle`, the computation of `Area` is likely to overflow with relatively small inputs.⁵ The code to check for this overflow is not placed immediately after the exception-prone statement; rather, it is separated and put into the `EXCEPTION` section of the block. Thus the flow of control for normal processing is clear and uninterrupted. If arithmetic overflow does happen, the system will raise the `Numeric_Error` exception flag and abort the block from the point of `* 22`. Control will pass directly to the statement `RAISE Diameter_Too_Big_Error`, then leave the enclosing block, which, in this case, is the body of the `One_Circle` procedure.

The command `RAISE Diameter_Too_Big_Error` renames the exception condition, an action which seems pointless in a short example. However, this can be an important way to communicate information. In a large program, there might be many situations in which a `Numeric_Error` could be generated, and many different potential responses. Renaming the exception provides more information to the receiver about the cause of the problem and the context in which it arose and makes it possible to take more specific corrective action.

The act of raising the new exception sends control back to procedure `Circles` with the information that the diameter was too large. This procedure takes corrective action by asking the user to reenter the data. At this point, the exception handling is complete. The exception has been

⁵On our system, integers are 2 bytes, and overflow occurs for `diameter = 205`.)

“caught”, and it is not passed up to the next level. The problem has no further repercussions; normal processing is resumed.

Exercises

1. What is spaghetti code? What causes it?
2. What are the three faults inherent in the `GOTO` control structure?
3. Why was `GOTO` initially used in the development of programming languages?
4. Name two programming languages in which any line of the program may be the target of a `GOTO`, whether or not that line has a declared statement label.
5. Name two programming languages that do not support `GOTO` at all.
6. What are the current arguments against using `GOTO`s? For using `GOTO`s?
7. What is a statement label? Why is it necessary?
8. What are the two ways to leave a loop? Give a specific example of each.
9. Explain the difference between `BREAK` and `GOTO`. Why is `BREAK` a superior control structure?
10. Explain why `Ada` has loop labels in addition to an `EXIT` statement.
11. Explain why labels in `FORTRAN` are more semantically sound than labels in `BASIC`.
12. Briefly, what is a continuation?
13. Why are continuations interesting?
14. Describe two applications of continuations.
15. How do exceptions arise? What is the result of ignoring them?
16. Compare exception handling in `Pascal` and in `C`.
17. Explain why exceptions are connected to nonlocal transfers of control.
18. Compare I/O error handling in `C` and in `Ada`. Comment on lexical coherence and human engineering.
19. Compare an exception name declaration in `Ada` to a label declaration in `Pascal`. Comment on purpose and likely implementation.
20. Why is an exception control structure superior to a `GOTO` for containing and recovering from errors?