Chapter 2

Control Structures

Assembler language only provides **goto** and its conditional variants. Early high-level languages such as FORTRAN relied heavily on **goto**, three-way arithmetic branches, and many-way indexed branches. Algol introduced control structures that began to make **goto** obsolete. Under the banner of "structured programming," computer scientists such as C. A. R. Hoare, Edsger W. Dijkstra, Donald E. Knuth, and Ole-Johan Dahl showed how programs could be written more clearly and elegantly with **while** and **for** loops, **case** statements, and loops with internal exits [Knuth 71; Dahl 72]. One of the tenets of structured programming is that procedures should be used heavily to modularize effort. In this chapter we will explore control structures that are a little out of the ordinary.

1 ◆ EXCEPTION HANDLING

If a procedure discovers that an erroneous situation (such as bad input) has arisen, it needs to report that fact to its caller. One way to program this behavior is to have each procedure provide an error return and to check for that return on each invocation. SNOBOL allows an explicit failure **goto** and success **goto** on each statement, which makes this sort of programming convenient. However, using a **goto** to deal with errors does not lead to clear programs, and checking each procedure invocation for error returns makes for verbose programs.

A control construct for dealing with error conditions was first proposed by Goodenough [Goodenough 75] and has found its way into languages like Ada, Mesa, CLU, ML, Eiffel, and Modula-3. I will use a syntax like Ada's for describing this control structure.

When a procedure needs to indicate failure, it **raises** an **exception**. This action causes control to transfer along a well-defined path in the program to where the exception is **handled**. To embed this concept in programming lan-

guages, identifiers can be declared to be of type **exception**. Each such identifier represents a distinct exception; the programmer usually names exception identifiers to indicate when they are to be raised, such as `StackOverflow`. Some built-in operations may raise exceptions on some arguments. For example, division by zero raises the predefined exception `DivByZero`. Converting an integer to a float in such a way that precision is lost might raise the exception `PrecisionLoss`. Trying to extract the head of an empty list might raise the exception `ListEmpty`.

A raised exception causes control to exit from the current expression, statement, and procedure, exiting outward until either the entire program is exited or control reaches a place where the program is explicitly prepared to handle the raised exception. For example:

Figure 2.1

```
variable                                                    1
     A, B : integer;                                        2
begin                                                       3
     B := 0;                                                4
     A := (4 / B) + 13;                                     5
     write(A);                                              6
handle                                                      7
     when DivByZero => A := 0;                              8
     when PrecisionLoss => B := 2;                          9
end;                                                        10
```

When control reaches line 5, a divide error occurs, raising `DivByZero`. Control exits from the expression (no addition of 13 occurs) and from the body of the block (line 6 is not executed). It would exit entirely from the block, but this block has a handler (lines 7– 9) that includes this particular exception (line 8). Control therefore continues on line 8, setting `A` to `0`. After that, the block exits (and `A` disappears, but let's ignore that.) If an exception had been raised that this block does not handle (even if it handles other exceptions), control would have continued to exit outward. If the raised exception causes the program to terminate, the runtime library might print a message indicating the exception name and a backtrace showing where the program was executing when the exception was raised.

It is also possible to associate an exception handler directly with an expression:

Figure 2.2

```
if ((A / B) handle when DivByZero => return 0) = 3          1
then ...                                                    2
```

Here I have used **return** instead of **do** to indicate that the handler yields a value to be used in the larger expression.

Languages that provide for exception handling usually allow the programmer to define new exceptions and explicitly raise them.

Figure 2.3

```
variable                                                    1
     BadInput : exception;                                  2
     A : integer;                                           3
```

```
begin                                                                    4
     read(A);                                                            5
     if A < 0 then                                                       6
          raise BadInput                                                 7
     end;                                                                8
     ...                                                                 9
handle                                                                   10
     when BadInput =>                                                    11
          write("Negative numbers are invalid here.");                   12
          raise BadInput;                                                13
end;                                                                     14
```

BadInput is a programmer-defined exception declared in line 2, raised in line 7, and handled in lines 11–13. This example also shows that a handler can reraise the same exception (or raise a different one) in order to propagate the raised exception further.

Perhaps I want all divide errors to yield 0 for the entire program. It is tedious to place a handler on each expression; instead, a language might allow execution to resume from a handler.

Figure 2.4

```
variable                                                                 1
     A, B : integer;                                                     2
begin                                                                    3
     B := 0;                                                             4
     A := (4 / B) + 13;                                                  5
     write(A);                                                           6
handle                                                                   7
     when DivByZero => resume 0;                                         8
end;                                                                     9
```

In this example, line 6 will be executed and will print 13. The DivByZero exception is raised in the middle of an expression, so it makes sense to resume the expression with a given value.

Unfortunately, resuming computation can be ill-defined. It is not always clear where to resume computation: at the point at which **raise** occurred or at some intermediate point along the exit path from that point to where the exception is handled. For example,

Figure 2.5

```
A := (GetInput() handle when BadInput => resume 0);
```

Does **resume** 0 mean that GetInput should return 0, or does it mean that computation should continue inside GetInput (perhaps at a **raise** statement, where 0 makes no sense)?

Luckily, programmers can usually manage quite well without needing to resume computation. A statement that might fail can be surrounded by a handler in a loop. If the statement fails, the handler can print diagnostic information, and the loop can try again.

Exceptions introduce several scope problems. First, the name scope that handles a raised exception generally has no access to the name scope that raised it. Therefore, there is no way for the handler to manipulate variables local to the raising scope in order to compute alternative answers or even to

generate error messages that convey exactly which values were erroneous. This problem is ameliorated in Modula-3, in which exceptions can take value-mode parameters. The actual parameters are provided by the **raise** statement, and the formal parameters are defined by the **handle** clause. Parameters can be used to indicate where in the program the exception was raised and what values led to the exceptional situation.

Second, programmer-defined exceptions may be visible in the raising scope but not in the handling scope. The problem arises for programmer-defined exceptions that exit the entire program (to a scope where only predefined exceptions exist) and for "don't-care" exception-handler patterns within the program, as in line 4 below:

Figure 2.6

```
begin                                          1
     ...                                       2
handle                                         3
     when _ => ...                             4
end;                                           5
```

Such a handler might not be able to raise the exception further (unless the programming language provides a predefined exception identifier `Self` that holds the exception that was raised).

In some ways, **raise** statements are like **goto** statements to labels passed as parameters. However, exceptions are far more disciplined than **goto**s, and they do not require that the programmer pass targets as parameters.

Exceptions reduce the clarity of loop constructs. Every loop has an implicit exit caused by an unhandled exception wresting control out of the loop. Modula-3 unifies loops and exceptions by treating **break** as equivalent to **raise** `ExitException`. Loop statements implicitly handle this exception and exit the loop. Similarly, Modula-3 considers the **return** statement as equivalent to **raise** `ReturnException`. The value returned by a function becomes the parameter to `ReturnException`.

The exception mechanism I have shown binds exception handlers to blocks. An alternative is to let raised exceptions throw the computation into a failure state [Wong 90]. In failure state, ordinary statements are not executed. Procedures can return while execution is in failure state, however. Only the **handle** statement is executed in failure state; after it completes, failure state is no longer in force unless **handle** reraises an exception. The programmer may place **handle** statements in the middle of blocks, interspersed with ordinary statements. The execution cost for this scheme may be fairly high, however, because every statement must be compiled with a test to see if execution is in failure state.

Exceptions are useful for more than handling error conditions. They also provide a clean way for programs to exit multiple procedure invocations. For example, an interactive editor might raise an exception in order to return to the main command loop after performing a complex action.

Exceptions are not the only reasonable way to handle error conditions. Sometimes it is easier for the programmer to have errors set a global variable that the program may inspect later when it is convenient. For example, the standard library packaged with C has a global variable `errno` that indicates the most recent error that occurred in performing an operating-system call.

The programmer can choose to ignore return values and inspect `errno` well into the calculation, redirecting further effort if an error has occurred. The program is likely to be more efficient and clearer than a program that surrounds code with exception handlers. This point is especially important in numerical computations on large data sets on highly pipelined computers. Putting in the necessary tests to handle exceptions can slow down such computations so much that they become useless, whereas hardware that sets a flag when it discovers overflow, say, allows such computations to run at full speed and lets the program notice rare problems after the fact.

Another way to treat errors is by generating **error values**, such as `undefined` and `positive_overflow`, that are an integral part of arithmetic types. Similarly, `null_pointer_dereference` and `array_range_error` can be error values generated by the related mistakes. Expressions can evaluate to an error value instead of their normal results. These error values are propagated (using specific rules) to produce a final result. For example, `1/0` yields the value `zero_divide`, while `0*(1/0)` yields `undefined`. Any operation involving `zero_divide` yields `undefined`. Error values render the results of all computations well defined, guaranteeing that all valid evaluation orders produce the same result.[1] They also provide for a degree of error repair, since the program can test for error values and perhaps transform them into something meaningful. However, because the program can continue computing with error values, the error values finally produced may provide no indication of the original errors. It can be quite difficult to debug programs when errors propagate in this way. It would be far more helpful if the error value contained extra information, such as the source file and line number where the error occurred, which could propagate along with the error value itself.

2 ◆ COROUTINES

Consider the problem of comparing two binary trees to see if their nodes have the same values in symmetric (also called in-order) traversal. For example, the trees in Figure 2.7 compare as equal.

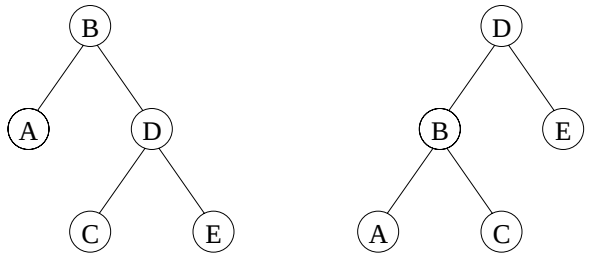

Figure 2.7 Equivalent binary trees

We could use a recursive procedure to store the symmetric-order traversal in an array, call the procedure for each tree, and then compare the arrays, but it is more elegant to advance independently in each tree, comparing as we go. Such an algorithm is also far more efficient if the trees are unequal near the

[1] An error algebra with good numeric properties is discussed in [Wetherell 83].

beginning of the traversal. The problem is that each traversal needs its own recursion stack. In most programming languages, this solution requires an explicit stack for each tree and a program that replaces recursion with iteration.

2.1 Coroutines in Simula

Simula provides explicit coroutines that have just the effect we need. Simula classes are introduced in Chapter 3 as a way to implement abstract data types. Here I will show you another use.

A class is a data type much like a record structure, but it may also contain procedures and initialization code. When a variable is declared of that class type, or when a value is created at runtime from the heap using a new call, an instance of the class is created. This instance is often called an **object**; in fact, the concept of object-oriented programming, discussed in Chapter 5, is derived largely from Simula classes. After space is allocated (either on the stack or the heap), the initialization code is run for this object. Programmers usually use this facility to set up the object's data fields. However, the initialization code may suspend itself before it completes. I will call an object that has not completed its initialization code an **active object**. An active object's fields may be inspected and modified, and its procedures may be called. In addition, its initialization code can be resumed from the point it suspended.

Because the initialization can invoke arbitrary procedures that may suspend at any point during their execution, each object needs its own stack until its initialization has completed. An active object is therefore a **coroutine**, that is, an execution thread that can pass control to other coroutines without losing its current execution environment, such as its location within nested name scopes and nested control structures.

Simula achieves this structure by introducing two new statements. The **call** statement specifies a suspended active object, which is thereby allowed to continue execution in its saved execution environment. The callers are saved on a runtime stack. The **detach** statement suspends the current object and returns control to the most recent object that invoked **call**. (This object is found on the stack just mentioned.) The main program is treated as an object for this purpose, but it must not invoke **detach**.

The program of Figure 2.8 solves the binary-tree equality puzzle. Simula syntax is fairly similar to Ada syntax; the following is close to correct Simula, although I have modified it somewhat so I don't confuse syntactic with semantic issues.

Figure 2.8

```
class Tree; -- used as a Pascal record                          1
     Value : char;                                              2
     LeftChild, RightChild : pointer to Tree;                   3
end; -- Tree                                                    4
```

```
class TreeSearch; -- used as a coroutine                          5
     MyTree : pointer to Tree;                                    6
     CurrentNode : pointer to Tree;                               7
     Done : Boolean; -- true when tree exhausted                  8

     procedure Dive                                               9
          (readonly Node : pointer to Tree);                      10
     begin                                                        11
          if Node ≠ nil then                                      12
               Dive(Node^.LeftChild);                             13
               CurrentNode := Node;                               14
               detach;                                            15
               Dive(Node^.RightChild);                            16
          end;                                                    17
     end; -- Dive                                                 18

begin -- TreeSearch: initialization and coroutine                 19
     Done := false;                                               20
     CurrentNode := nil;                                          21
     detach; -- wait for initial values                           22
     Dive(MyTree); -- will detach at each node                    23
     Done := true;                                                24
end; -- TreeSearch                                                25

variable -- main                                                   26
     A, B : pointer to Tree;                                      27
     ASearch, BSearch : pointer to TreeSearch;                    28
     Equal : Boolean;                                             29
begin -- main                                                      30
     ... -- initialize A and B                                    31
     new(ASearch); ASearch^.MyTree := A;                          32
     new(BSearch); BSearch^.MyTree := B;                          33
     while not (ASearch^.Done or BSearch^.Done or                 34
          ASearch^.CurrentNode ≠ BSearch^.CurrentNode)            35
     do                                                           36
          call ASearch^; -- continues coroutine                   37
          call BSearch^; -- continues coroutine                   38
     end;                                                         39
     Equal := ASearch^.Done and BSearch^.Done;                    40
end;                                                               41
```

The new calls in lines 32–33 create new instances of TreeSearch and assign them to ASearch and BSearch. Each of these instances detaches during initialization (line 22) to allow their local variables MyTree to be set (lines 32–33). Then they are repeatedly resumed by the main program (lines 37–38). The **call** statements in lines 37–38 are invalid after the coroutines have finished (that is, after the initialization code of the class instances ASearch^ and BSearch^ has finished line 24), but line 34 prevents such a mistake from occurring. The class instances for both the trees and the coroutines are deallocated after control exits from the block at line 41, since all pointers to those instances disappear at that point. (Garbage collection is used for deallocation.)

2.2 Coroutines in CLU

The CLU language, designed by Barbara Liskov at MIT, provides a generalized **for** loop [Liskov 81]. The control variable takes on successive values provided by a coroutine called an **iterator**. This iterator is similar in most ways to an ordinary procedure, but it returns values via a **yield** statement. When the **for** loop requires another value for the control variable, the iterator is resumed from where it left off and is allowed to execute until it encounters another **yield**. If the iterator reaches the end of its code instead, the **for** loop that relies on the iterator terminates. CLU's **yield** is like Simula's **detach**, except that it also passes back a value. CLU's **for** implicitly contains the effect of Simula's **call**.

A naive implementation of CLU would create a separate stack for each active iterator instance. (The same iterator may have several active instances; it does, for example, if there is a **for** nested within another **for**.) A coroutine linkage, much like Simula's **call** and **detach**, would ensure that each iterator instance maintains its own context, so that it may be resumed properly.

The following program provides a simple example. CLU syntax is also fairly close to Ada syntax; the following is almost valid CLU.

Figure 2.9

```
iterator B() : integer; -- yields 3, 4                          1
begin                                                           2
     yield 3;                                                   3
     yield 4;                                                   4
end; -- B                                                       5

iterator C() : integer; -- yields 1, 2, 3                       6
begin                                                           7
     yield 1;                                                   8
     yield 2;                                                   9
     yield 3;                                                   10
end; -- C                                                       11

iterator A() : integer; -- yields 10, 20, 30                    12
variable                                                        13
     Answer : integer;                                          14
begin                                                           15
     for Answer := C() do -- ranges over 1, 2, 3                16
          yield 10*Answer;                                      17
     end;                                                       18
end; -- A                                                       19

variable                                                        20
     x, y : integer;                                            21
begin                                                           22
     for x := A() do -- ranges over 10, 20, 30                  23
          for y := B() do -- ranges over 3, 4                   24
               P(x, y); -- called 6 times                       25
          end;                                                  26
     end;                                                       27
end;                                                            28
```

The loop in line 23 iterates over the three values yielded by iterator A (lines

12–19). For each of these values, the loop in line 24 iterates over the two values yielded by iterator B (lines 1–5). Iterator A itself introduces a loop that iterates over the three values yielded by iterator C (lines 6–11).

Happily, CLU can be implemented with a single stack. As a **for** loop begins execution, some activation record (call it the *parent*) is active (although not necessarily at the top of the stack). A new activation record for the iterator is constructed and placed at the top of the stack. Whenever the body of the loop is executing, the parent activation record is current, even though the iterator's activation record is higher on the stack. When the iterator is resumed so that it can produce the next value for the control variable, its activation record again becomes current. Each new iterator invocation gets a new activation record at the current stack top. Thus an activation record fairly deep in the stack can be the parent of an activation record at the top of the stack. Nonetheless, when an iterator terminates, indicating to its parent **for** loop that there are no more values, the iterator's activation record is certain to be at the top of the stack and may be reclaimed by simply adjusting the top-of-stack pointer. (This claim is addressed in Exercise 2.10.)

For Figure 2.9, each time P is invoked, the runtime stack appears as follows. The arrows show the dynamic (child-parent) chain.

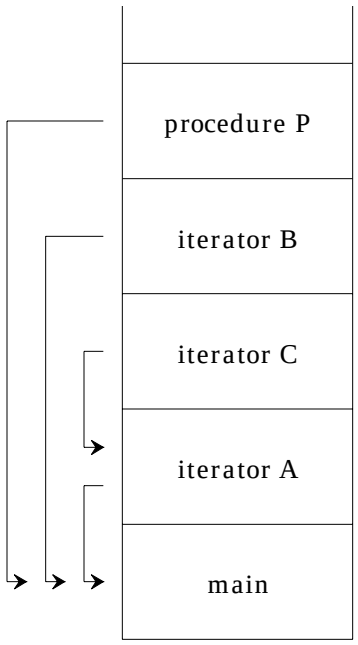

Figure 2.10 Runtime CLU stack during iterator execution

CLU iterators are often trivially equivalent to programs using ordinary **for** loops. However, for some combinatorial algorithms, recursive CLU iterators are much more powerful and allow truly elegant programs. One example is the generation of all binary trees with *n* nodes. This problem can be solved without CLU iterators, albeit with some complexity [Solomon 80]. Figure 2.11 presents a natural CLU implementation.

Figure 2.11

```
type Tree =                                                          1
     record                                                          2
          Left, Right : pointer to Node;                             3
     end; -- Tree;                                                   4

iterator TreeGen(Size : integer) : pointer to Tree;                  5
-- generate all trees with Size nodes                                6
variable                                                             7
     Answer : Tree;                                                  8
     Root : integer; -- serial number of the root                    9
begin                                                                10
     if Size = 0 then                                                11
          yield nil; -- only the empty tree                          12
     else -- answer not empty                                        13
          for Root := 1 to Size do                                   14
               for Answer.Left := TreeGen(Root-1) do                 15
                    for Answer.Right := TreeGen(Size-Root)           16
                    do                                               17
                         yield &Answer;                              18
                    end; -- for Right                                19
               end; -- for Left                                      20
          end; -- for Root                                           21
     end -- answer not empty                                         22
end -- TreeGen                                                       23

variable -- sample use of TreeGen                                    24
     T : pointer to Tree;                                            25
begin                                                                26
     for T := TreeGen(10) do                                         27
          TreePrint(T);                                              28
     end;                                                            29
end;                                                                 30
```

This marvelously compact program prints all binary trees of size 10. The **for** loop in lines 27–29 invokes the iterator `TreeGen(10)` until no more values are produced. `TreeGen` will produce 16,796 values before it terminates. It works by recursion on the size of tree required. The simple case is to generate a tree of size 0; the **yield** in line 12 accomplishes this. If an instance of `TreeGen(0)` is resumed after line 12, it falls through, thereby terminating its parent loop. The other case requires that `TreeGen` iterate through all possibilities of the root of the tree it will generate (line 14). Any one of the `Size` nodes could be root. For each such possibility, there are `Root-1` nodes on the left and `Size-Root` nodes on the right. All combinations of the trees meeting these specifications must be joined to produce the trees with `Size` nodes. The nested loops starting in lines 15 and 16 iterate through all such combinations; for each, **yield** in line 18 passes to the parent a reference to the solution. The storage for the solution is in the local activation record of the iterator. As iterators terminate, their storage is released, so there is no need to explicitly allocate or deallocate any storage for the resulting tree.

2.3 Embedding CLU Iterators in C

Surprisingly, it is possible to implement CLU iterators using only the constructs available to a C programmer. This implementation clarifies CLU and shows some interesting aspects of C.

The only machine-independent way to manipulate activation records in C is to use the library routines `setjmp` and `longjmp`. They are intended to provide the equivalent of exception handling; they allow many levels of activation records to be terminated at once, jumping from an activation record at the top of the stack directly back to one deep within the stack. I apply these routines in a way probably unintended by their inventors: to resume an activation record higher on the stack than the invoker.

`Setjmp(Buf)` takes a snapshot of the current environment — registers, stack pointers, program counter, and such — and places it in the `Buf` data structure. `Longjmp(Buf, ReturnValue)` restores the registers from `Buf`, effectively restoring the exact context in which the `setjmp` was called. In fact, it creates another return from the original `setjmp` call. In order to let the program distinguish whether `setjmp` is returning the ordinary way or because of a `longjmp`, `setjmp` returns a 0 in the former case and `ReturnValue` in the latter case. For this reason, `setjmp` is usually embedded in a conditional or case statement to identify these cases and take appropriate action.

This facility is very like jumping to a label passed as a parameter, which has the effect of unwinding the stack to the right activation record for the target of the **`goto`**. `Setjmp` can capture the situation before a procedure call, and `longjmp` can be invoked from within a procedure; the call unwinds the stack to its position when `setjmp` recorded the situation. Unbridled use of `setjmp` and `longjmp` can be worse than an unconstrained **`goto`**. It allows such activities as jumping into a control structure (after all, the `setjmp` can be in the middle of a loop or a branch of a conditional) or even jumping back to a procedure that has exited.

This ability to break the rules makes it possible to implement CLU iterators within the C language. My implementation is packaged as a set of C macros, primarily `iterFOR` and `iterYIELD`. Whenever `iterFOR` is about to invoke an iterator, it performs `setjmp` to allow the iterator to come back to the `iterFOR` via `longjmp`. Likewise, each `iterYIELD` performs `setjmp` to allow its parent `iterFOR` to resume it via `longjmp`. The macros use a single global variable (not visible to the programmer) to store a pointer to the associated `Buf` structures in both these cases.

Now that the linkage between `iterFOR` and its iterator can be established, two problems remain. They both concern managing space on the stack. Unfortunately, new activation records are placed on the stack immediately above the invoker's activation record, even if other activation records have been placed there.

The first problem is that even in the simplest situation, with a single `iterFOR` invoking a single iterator, we need padding on the stack between their respective activation records. If there is no padding, then attempts by `iterFOR` to resume the iterator fail. After all, `iterFOR` calls `longjmp`, and this invocation places an activation record on the stack (since `longjmp` is also a procedure). This activation record coincides with the iterator's activation record, destroying at least the arguments and quite likely other information as well. Furthermore, any ordinary procedure calls invoked by the body of

the iterFOR need a place to put their activation records. I solve this problem by invoking iterators via a Helper routine, which declares a local array just for padding and then calls the iterator by a normal procedure invocation.

The second problem arises with nested iterFOR loops, which are, after all, the interesting ones. Consider again Figure 2.9 introduced on page 34. Once the outer **for** in line 23 has established an instance of A, and A in line 16 has established an instance of C, the inner **for** in line 24 needs to put its instance of B at the top of the stack. Main can't directly invoke Helper, because that would place the activation record for B exactly where the A is residing. I therefore keep track of the iterator instance (in this case, C) that is currently at the top of the stack so that I can resume it, not so it will yield its next value, but so it will call Helper on my behalf to start B.

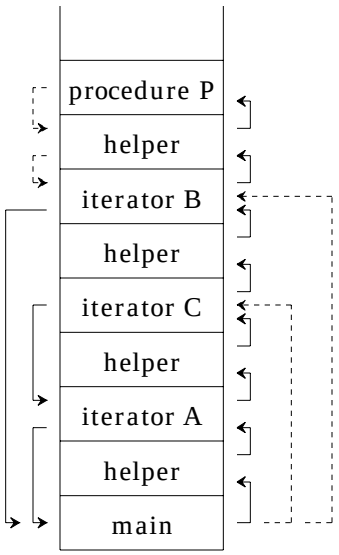

Figure 2.12 Runtime C stack during iterator execution

Figure 2.12 demonstrates the appearance of the runtime stack at the same stage as the previous figure. Solid arrows pointing downward show to which activation record each activation record returns control via longjmp. Dotted arrows pointing downward show ordinary procedure returns. Solid arrows pointing upward show which activation record actually started each new activation record. Dotted arrows pointing upward show the direction of longjmp used to request new invocations.

Choosing to build iterators as C macros provides the ability to express CLU coroutines at the cost of clarity. In particular, I made the following decisions:

1. The value returned by an iterYIELD statement must be placed in a global variable by the programmer; the macros do not attempt to transmit these values.
2. The programmer must write the Helper routine. In the usual case, the helper just declares a dummy local array and invokes the iterator procedure, passing arguments through global variables. If there are several different iterators to be called, Helper must distinguish which one is intended.

3. Any routine that includes an iterFOR and every iterator must invoke iterSTART at the end of local declarations.
4. Instead of falling through, iterators must terminate with iterDONE.
5. The Helper routine does not provide enough padding to allow iterators and their callers to invoke arbitrary subroutines while iterators are on the stack. Procedures must be invoked from inside iterFOR loops by calling iterSUB.

The macro package appears in Figure 2.13. (Note for the reader unfamiliar with C: the braces { and } act as **begin** and **end**; void is a type with no values; declarations first give the type (such as int or jmp_buf *) and then the identifier; the assignment operator is = ; the dereferencing operator is * ; the referencing operator is & .)

Figure 2.13

```
#include <setjmp.h>                                                    1
#define ITERMAXDEPTH 50                                                2
jmp_buf    *GlobalJmpBuf;  /* global pointer for linkage */            3
jmp_buf    *EnvironmentStack[ITERMAXDEPTH] = {0},                      4
     **LastEnv = EnvironmentStack;                                     5

/* return values for longjmp */                                       6
#define J_FIRST 0 /* original return from setjmp */                    7
#define J_YIELD 1                                                      8
#define J_RESUME 2                                                     9
#define J_CALLITER 3                                                   10
#define J_DONE 4                                                       11
#define J_CALLSUB 5                                                    12
#define J_RETURN 6                                                     13

/*   iterSTART must be invoked after all local declarations           14
     in any procedure with an iterFOR and in all iterators.           15
*/                                                                     16
#define iterSTART \                                                    17
     jmp_buf MyBuf, CallerBuf; \                                       18
     if (GlobalJmpBuf) \                                               19
          bcopy((char *)GlobalJmpBuf, (char *)CallerBuf, \             20
               sizeof(jmp_buf)); \                                     21
     LastEnv++; \                                                      22
     *LastEnv = &MyBuf;                                                23
```

```
/*   Initialization gives global args to Helper.
     Body is the body of the for loop.
*/
#define iterFOR(Initialization, Body) \
     switch (setjmp(MyBuf)) { \
          case J_FIRST: \
               GlobalJmpBuf = MyBuf; \
               Initialization; \
               if (*LastEnv != MyBuf)\
                    longjmp(**LastEnv, J_CALLITER); \
               else Helper(); \
          case J_YIELD: \
               {    jmp_buf *Resume = GlobalJmpBuf; \
                    Body; \
                    longjmp(*Resume, J_RESUME); \
               } \
          case J_DONE: break; \
     }

/*   No arguments; the value yielded must be passed
     through globals.
*/
#define iterYIELD \
     switch (setjmp(MyBuf)) { \
          case J_FIRST: \
               GlobalJmpBuf = &MyBuf; \
               longjmp(CallerBuf, J_YIELD); \
          case J_CALLITER: \
               Helper();  /* won't return */ \
          case J_CALLSUB: \
               {    jmp_buf *Return = GlobalJmpBuf; \
                    Helper(); \
                    longjmp(*Return, J_RETURN); \
               } \
          case J_RESUME: break; \
     }

/*   Every iterator must return via iterDONE;
     a direct return is meaningless.
*/
#define iterDONE \
     LastEnv--; \
     longjmp(CallerBuf, J_DONE)
```

```
/*   iterSUB(Initialization) invokes Helper to perform
     subroutine work from an iterator or its user.
*/
#define iterSUB(Initialization) \
     {    jmp_buf SubBuf; \
          switch (setjmp(SubBuf)) { \
               case J_FIRST: \
                    Initialization; \
                    if (*LastEnv != &MyBuf) { \
                         GlobalJmpBuf = &SubBuf; \
                         longjmp(**LastEnv, J_CALLSUB); \
                    } \
                    else Helper(); \
                    break; \
               case J_RETURN: \
                    break; \
          } \
     }
```

The variables used to remember the stack of environments are `EnvironmentStack` and `LastEnv` (lines 4 and 5). When an iterator starts, it must save a copy of its parent's `Buf` (lines 20–21); this code is in a conditional, since `iterStart` is also called by noniterators that happen to invoke iterators. An iterator is invoked through `Helper` (line 34) or by asking a more deeply nested iterator to assist (line 33). Such calls for assistance always appear as resumptions from `iterYIELD` (line 50).

`iterSUB` (line 68) invokes `Helper` from the top of the stack but expects a normal return. `Helper` needs to be able to identify which subroutine is actually to be called by inspecting global variables. The flow of control travels to the top of the stack (line 75), where it invokes `Helper` (line 54) and then returns via a `longjmp` (line 55).

Figure 2.14 shows how to code Figure 2.9 (on page 34) using the C macros.

Figure 2.14

```
int AValue, BValue, CValue, mainX, mainY;
enum {CallA, CallB, CallC, CallPrint} HelpBy;

void Helper(){
     switch (HelpBy) {
          case CallA: A(); break;
          case CallB: B(); break;
          case CallC: C(); break;
          case CallPrint:
               printf("%d %d0, mainX, mainY); break;
     }
}

int B(){
     iterSTART;
     BValue = 3; iterYIELD;
     BValue = 4; iterYIELD;
     iterDONE;
}
```

```
int C(){                                                              18
     iterSTART;                                                       19
     CValue = 1; iterYIELD;                                           20
     CValue = 2; iterYIELD;                                           21
     CValue = 3; iterYIELD;                                           22
     iterDONE;                                                        23
}                                                                     24

int A(){                                                              25
     int Answer;                                                      26
     iterSTART;                                                       27
     iterFOR ({HelpBy = CallC;} , {                                   28
          Answer = 10 * CValue;                                       29
          AValue = Answer; iterYIELD;                                 30
     });                                                              31
     iterDONE;                                                        32
}                                                                     33

void main(){                                                          34
     iterSTART;                                                       35
     iterFOR({HelpBy = CallA;} , {                                    36
          mainX = AValue;                                             37
          iterFOR ({HelpBy = CallB;} , {                              38
               mainY = BValue;                                        39
               iterSUB( HelpBy = CallPrint );                         40
          });                                                         41
     });                                                              42
}                                                                     43
```

Line 1 introduces all the variables that need to be passed as parameters or as results of **`yield`** statements. Lines 2–11 form the `Helper` routine that is needed for invoking iterators as well as other routines, such as `printf`.

I cannot entirely recommend using these C macros; it is far better to use a language that provides iterators directly for those situations (admittedly rare) when recursive iterators are the best tool. After all, CLU iterators are not at all hard to compile into fine code.

The C macros can be used (I have used them on several occasions), but they leave a lot of room for errors. The programmer must pass parameters and results to and from the iterators through global variables. All calls to iterators (via `iterFOR`) and to routines (via `iterSUB`) are funneled through a single `Helper` routine. `Helper` needs to reserve adequate space (experience shows that not much is needed) and must use global variables to distinguish the reason it is being called. The programmer must be careful to use `iterSUB` instead of direct calls inside `iterFOR`. The resulting programs are certainly not elegant in appearance, although with some practice, they are not hard to code and to read.

The C macros have other drawbacks. In some C implementations, `longjmp` refuses to jump up the stack. Compile-time and hand-coded optimizations that put variables in registers typically render them invisible to `setjmp`, so iterators and routines that contain `iterFOR` must not be optimized. There is a danger that interrupts may cause the stack to become garbled, because a program written in C cannot protect the top of the stack.

2.4 Coroutines in Icon

Icon is discussed in some detail in Chapter 9. It generalizes CLU iterators by providing expressions that can be reevaluated to give different results.

3 ◆ CONTINUATIONS: IO

FORTRAN demonstrates that is possible to build a perfectly usable programming language with only procedure calls and conditional **goto** as control structures. The Io language reflects the hope that a usable programming language can result from only a single control structure: a **goto** with parameters. I will call the targets of these jumps procedures even though they do not return to the calling point. The parameters passed to procedures are not restricted to simple values. They may also be **continuations**, which represent the remainder of the computation to be performed after the called procedure is finished with its other work. Instead of returning, procedures just invoke their continuation. Continuations are explored formally in Chapter 10; here I will show you a practical use.

Io manages to build remarkably sophisticated facilities on such a simple foundation. It can form data structures by embedding them in procedures, and it can represent coroutines.

Io programs do not contain a sequence of statements. A program is a procedure call that is given the rest of the program as a continuation parameter. A statement continuation is a closure; it includes a procedure, its environment, and even its parameters.

Io's syntax is designed to make statement continuations easy to write. If a statement continuation is the last parameter, which is the usual case, it is separated from the other parameters by a semicolon, to remind the programmer of sequencing. Continuations and procedures in other parameter positions must be surrounded by parentheses. I will present Io by showing examples from [Levien 89].

Figure 2.15

```
write 5;                                                    1
write 6;                                                    2
terminate                                                   3
```

As you expect, this program prints 5 6. But I need to explain how it works. The predeclared `write` procedure takes two parameters: a number and a continuation. The call in line 1 has 5 as its first parameter and `write 6; terminate` as its second. The `write` procedure prints 5 and then invokes the continuation. It is a call to another instance of `write` (line 2), with parameters 6 and `terminate`. This instance prints 6 and then invokes the parameterless predeclared procedure `terminate`. This procedure does nothing. It certainly doesn't return, and it has no continuation to invoke.

Procedures can be declared as follows:

Figure 2.16

```
declare writeTwice: → Number;                               1
     write Number; write Number; terminate.                 2
```

That is, the identifier `writeTwice` is associated with an anonymous procedure

(introduced by →) that takes a single formal parameter Number (the parameter list is terminated by the first ;) and prints it twice. The period, . , indicates the end of the declaration. This procedure is not very useful, because execution will halt after it finishes. Procedures do not return. So I will modify it to take a continuation as well:

Figure 2.17

```
declare writeTwice: → Number Continuation;                    1
     write Number; write Number; Continuation.                2

writeTwice 7;                                                 3
write 9;                                                      4
terminate                                                     5
```

Lines 1–2 declare writeTwice, and line 3 invokes it with a 7 and a continuation composed of lines 4–5. Here is a trace of execution:

Figure 2.18

```
writeTwice 7 (write 9; terminate) -- called on line 3         1
     Number := 7                                              2
     Continuation := (write 9; terminate)                     3
write 7 (write 7; write 9; terminate) -- called on line 2     4
     -- writes 7                                              5
write 7 (write 9; terminate) -- called by write               6
     -- writes 7                                              7
write 9 (terminate) -- called by write                        8
     -- writes 9                                              9
terminate  -- called by write                                 10
```

Indented lines (such as lines 2–3) indicate the formal-actual bindings. I surround parameters in parentheses for clarity.

Even arithmetic operations are built to take a continuation. The difference between a statement and an expression is that an expression continuation expects a parameter, namely, the value of the expression. Consider the following code, for example:

Figure 2.19

```
+ 2 3 → Number;                                               1
write Number;                                                 2
terminate                                                     3
```

The + operator adds its parameters 2 and 3 and passes the resulting value 5 to its last parameter (→ Number; write Number; terminate), which prints the 5 and terminates. This expression continuation is an anonymous procedure; that is, it is declared but not associated with an identifier. In general, an expression continuation is a procedure expecting a single parameter. The syntax conspires to make this program look almost normal. The result of the addition is apparently assigned to a variable Number, which is used in the following statements. In fact, the result of the addition is bound to the formal parameter Number, whose scope continues to the end of the program.

Conditional operators are predeclared to take two statement continuations corresponding to the two Boolean values true and false. For example, the following code will print the numbers from 1 to 10.

Figure 2.20

```
declare Count: → Start End Continuation;                    1
     write Start;                                           2
     = Start End (Continuation); -- "then" clause           3
     + Start 1 → NewStart; -- "else clause"                 4
     Count NewStart End Continuation.                       5
Count 1 10; terminate                                       6
```

Here is a trace of execution:

Figure 2.21

```
Count 1 10 terminate -- called on line 6                    1
     Start := 1                                             2
     End := 10                                              3
     Continuation := terminate                              4
write 1 (= 1 10 terminate A:(+ 1 1 → NewStart;              5
     Count NewStart 10; terminate)                          6
     -- writes 1                                            7
= 1 10 terminate A                                          8
     -- called by write                                     9
A -- called by '='                                          10
+ 1 1 B:(→ NewStart; Count NewStart 10; terminate)          11
B 2 -- called by '+'                                        12
Count 2 10 terminate -- called by B                         13
...
Count 10 10 terminate                                       14
write 10 (= 10 10 terminate C:(                             15
     + 1 1 → NewStart; Count NewStart 10; terminate.)       16
     -- writes 10                                           17
= 10 10 terminate C                                         18
terminate                                                   19
```

I have introduced the shorthand forms A (line 5), B (line 11), and C (line 15) for conciseness.

Procedures can contain constants that are made available later:

Figure 2.22

```
declare TwoNumbers: → Client;                               1
     Client 34 53.                                          2
declare WritePair: → PairProc Continuation;                 3
     PairProc → x y;                                        4
     write x;                                               5
     write y;                                               6
     Continuation.                                          7
WritePair TwoNumbers;                                       8
terminate                                                   9
```

Line 8 invokes WritePair with two parameters: the first is a procedure (twoNumbers), and the second is a continuation (terminate). WritePair invokes its first parameter (line 4), passing the remainder of its body (lines 4–7) as a procedure parameter with local variable Continuation bound to terminate. TwoNumbers applies that procedure to parameters 34 and 53, causing these numbers to be printed and then terminate to be called. Procedure TwoNumbers can be generalized to contain any two numbers:

Figure 2.23

```
declare MakePair: → x y User Continuation;                    1
     User (→ Client; Client x y); Continuation.               2

MakePair 12 13 (WritePair);                                   3
terminate.                                                    4
```

The execution trace is as follows:

Figure 2.24

```
MakePair 12 13 WritePair terminate                            1
     x := 12                                                  2
     y := 14                                                  3
     User := WritePair                                        4
     Continuation := terminate                                5
WritePair A:(→ Client; Client 12 13) terminate                6
     PairProc := A                                            7
     Continuation := terminate                                8
A B:(→ x y; write x; write y; terminate);                     9
     Client := G                                              10
B 12 13 -- writes "12 13" then terminates.                    11
```

Linked lists can be implemented by suitable cleverness as functions with two parameters, both procedures. An empty list calls its first parameter, which is a continuation. Other lists call the second parameter, passing two new parameters that represent the first number in the list and the remainder of the list. Here are the relevant declarations:

Figure 2.25

```
declare WriteList: → List Continuation;                       1
     List (Continuation) → First Rest;                        2
     write First;                                             3
     WriteList Rest;                                          4
     Continuation.                                            5

declare EmptyList: → Null NotNull;                            6
     Null.                                                    7

declare Cons: → Number List EContinuation;                    8
     EContinuation → Null NotNull;                            9
     NotNull Number List.                                     10

Cons 1 EmptyList → List;                                      11
Cons 2 List → List;                                           12
WriteList List;                                               13
terminate                                                     14
```

Here, Cons (the name is taken from LISP, described in Chapter 4) is meant to combine a header element with the rest of a list to create a new list. Again, the execution trace clarifies what happens:

Figure 2.26

```
Cons 1 EmptyList A:(                                                    1
     → List; Cons 2 List → List; WriteList List;                        2
          terminate)                                                    3
     Number := 1                                                        4
     List := EmptyList                                                  5
     EContinuation := A                                                 6
A B:(→ Null NotNull; NotNull 1 EmptyList)                               7
     List := B                                                          8
Cons 2 B C:(→ List; WriteList List; terminate)                          9
     Number := 2                                                        10
     List := B                                                          11
     EContinuation := C                                                 12
C D:(→ Null NotNull; NotNull 2 B)                                       13
     List := D                                                          14
WriteList D terminate                                                   15
     List := D                                                          16
     Continuation := terminate                                          17
D terminate E:(→ First Rest; write First;                               18
     WriteList Rest; terminate)                                         19
     Null := terminate                                                  20
     NotNull := E                                                       21
E 2 B                                                                   22
     First := 2                                                         23
     Rest := B                                                          24
     -- writes 2                                                        25
WriteList B terminate                                                   26
     List := B                                                          27
     Continuation := terminate                                          28
B terminate F:(→ First Rest; write First;                               29
     WriteList Rest; terminate)                                         30
     Null := terminate                                                  31
     NotNull := F                                                       32
F 1 EmptyList                                                           33
     First := 1                                                         34
     Rest := EmptyList                                                  35
     -- writes 1                                                        36
WriteList EmptyList terminate                                           37
     List := EmptyList                                                  38
     Continuation := terminate                                          39
EmptyList terminate G:(→ First Rest; write First;                       40
     WriteList Rest; terminate)                                         41
     Null := terminate                                                  42
     NotNull := G                                                       43
terminate                                                               44
```

Similar cleverness can produce a set of declarations for binary trees. Empty trees call their first parameter. Other trees call their second parameter with the key, left subtree, and the right subtree. Other data structures can be built similarly.

Continuations are perfectly capable of handling coroutines. For example, a global variable could hold the continuation of the thread that is not currently executing. I could define a `switch` procedure that saves its continuation parameter in the global and invokes the old value of the global. It would be more elegant to redesign statement continuations. Instead of being a sin-

gle closure, they could be a list of closures (using the list mechanisms I have already introduced). The `switch` procedure would take the appropriate element from the list and sort it to the front of the list. Ordinary procedures use the front of the list as the current thread.

Instead of showing the gory details of coroutines, I will show how Io can build infinite data structures that are evaluated only when necessary. (Lazy evaluation is discussed in Chapter 4.)

Figure 2.27

```
declare Range: → First EContinuation;                    1
     EContinuation First → Null NotNull;                 2
     + First 1 → NewFirst;                               3
     Range NewFirst EContinuation.                       4
declare FullRange: → Null NotNull;                       5
     Range 0 NotNull.                                    6

WriteList FullRange; -- writes 0 1 2 3 ...               7
terminate                                                8
```

I leave it to you as an exercise to trace the execution.

Given that continuations are very powerful, why are they not a part of every language? Why do they not replace the conventional mechanisms of control structure? First, continuations are extremely confusing. The examples given in this section are almost impossible to understand without tracing, and even then, the general flow of control is lost in the details of procedure calls and parameter passing. With experience, programmers might become comfortable with them; however, continuations are so similar to **goto**s (with the added complexity of parameters) that they make it difficult to structure programs.

Second, continuations are not necessarily pleasant to implement. Procedures may be referenced long after they are created, and allocation does not follow a stack discipline, so it appears that activation records must be created in the heap. Luckily, circularities will not exist, so reference counts can govern reclamation of activation records. The implementation and the programmer must be able to distinguish functions that have not yet been bound to parameters (classical closures) from those that are so bound. Both are present in Io. In Figure 2.25 (on page 46), the anonymous procedure in lines 2–5 is a classical closure, whereas the subsidiary call to `write` in line 3 includes its parameters (`First` and `WriteList Rest; Continuation`).

Even though continuations will never be a popular programming method, I like them because they combine several ideas you will see elsewhere in this book. The examples abound with higher-level functions (discussed in Chapter 3) and anonymous functions (also Chapter 3). Continuations can implement coroutines and LISP-style lists (Chapter 4). Finally, denotational semantic definitions of programming languages use continuations directly (Chapter 10).

4 ◆ POWER LOOPS

Although the programmer usually knows exactly how deeply loops must nest, there are some problems for which the depth of nesting depends on the data. Programmers usually turn to recursion to handle these cases; each level of nesting is a new level of recursion. However, there is a clearer alternative that can generate faster code. The alternative has recently[2] been called **power loops** [Mandl 90]. The idea is to have an array of control variables and to build a loop that iterates over all control variables.

For example, the *n*-queens problem is to find all solutions to the puzzle of placing *n* queens on an $n \times n$ chessboard so that no queen attacks any other. Here is a straightforward solution:

Figure 2.28

```
variable                                                    1
     Queen : array 1 .. n of integer;                       2

nest Column := 1 to n                                       3
     for Queen[Column] := 1 to n do                         4
          if OkSoFar(Column) then                           5
               deeper;                                      6
          end; -- if OkSoFar(Column)                        7
     end; -- for Queen[Column]                              8
do                                                          9
     write(Queen[1..n]);                                    10
end;                                                        11
```

Any solution will have exactly one queen in each column of the chessboard. Line 2 establishes an array that will describe which row is occupied by the queen in each column. The `OkSoFar` routine (line 5) checks to make sure that the most recent queen does not attack (and therefore is not attacked by) any of the previously placed queens. Line 3 introduces a set of nested loops. It effectively replicates lines 4–8 for each value of `Column`, placing the next replica at the point marked by the **deeper** pseudostatement (line 6). There must be exactly one **deeper** in a **nest**. Nested inside the innermost instance is the body shown in line 10. If `n = 3`, for example, this program is equivalent to the code of Figure 2.29.

[2] The Madcap language had power loops in the early 1960s [Wells 63].

Figure 2.29

```
for Queen[1] := 1 to n do                                          1
     if OkSoFar(1) then                                            2
          for Queen[2] := 1 to n do                                3
               if OkSoFar(2) then                                  4
                    for Queen[3] := 1 to n do                      5
                         if OkSoFar(3) then                        6
                              write(Queen[1..3])                   7
                         end; -- if 3                              8
                    end; -- for 3                                  9
               end; -- if 2                                        10
          end; -- for 2                                            11
     end; -- if 1                                                  12
end; -- for 1                                                      13
```

Nesting applies not only to loops, as Figure 2.30 shows.

Figure 2.30

```
nest Level := 1 to n                                               1
     if SomeCondition(Level) then                                  2
          deeper;                                                  3
     else                                                          4
          write("failed at level", Level);                         5
     end;                                                          6
do                                                                 7
     write("success!");                                            8
end;                                                               9
```

Of course, a programmer may place a **nest** inside another **nest**, either in the replicated part (as in lines 2–6 of Figure 2.30) or in the body (line 8), but such usage is likely to be confusing. If **nest** can be nested in the replicated part, each **deeper** must indicate which **nest** it refers to.

It is not hard to generate efficient code for **nest**. Figure 2.31 is a flowchart showing the generated code, where `i` is the **nest** control variable. The labels t and f are the `true` and `false` exits of the conditionals. Label d is the exit from the replicated part when it encounters **deeper**, and r is the reentry after **deeper**. The fall-through exit from the replicated part is called e. If execution after **deeper** will just fall through (as in Figure 2.30), decrementing `i` and checking `i < init` can be omitted.

Although power loops are elegant, they are subsumed by recursive procedures, albeit with a loss of elegance and efficiency. Power loops are so rarely helpful that languages should probably avoid them. It doesn't make sense to introduce a construct in a general-purpose language if it will only be used in a handful of programs.

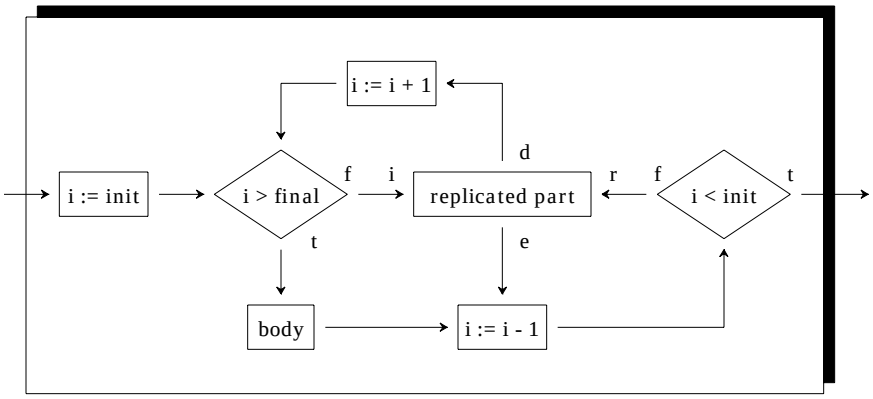

Figure 2.31 Flowchart for **nest**

5 ◆ FINAL COMMENTS

This chapter has introduced a variety of control constructs that have a mixed history of success. Exception-handling mechanisms are enjoying increasing popularity. General coroutines are found to some extent in concurrent programming languages (discussed in Chapter 7). CLU iterators and power loops never caught on in mainstream languages. Io continuations have no track record, but appear unlikely to catch on.

We can often see good reason for these results. Chapter 1 presented a list of characteristics of good programming languages. Among them were simplicity (using as few concepts as possible), clarity (easily understood code semantics), and expressiveness (ability to describe algorithms). Exception handling scores well on all these fronts. The mechanism introduces only one additional concept (the exception, with the **raise** statement and the **handle** syntax). The semantics are clear when an exception is raised, especially if no resumption is possible and if all exceptions must be declared at the global level. The only confusion might come from the fact that the handler is determined dynamically, not statically; dynamic binding tends to be more confusing to the programmer, because it cannot easily be localized to any region of the program. The mechanism serves a real need in expressing multilevel premature procedure return.

Coroutines are less successful by my measures. The set of concepts is not too large; Simula manages with per-object initialization code and two new statements: **detach** and **call**. However, the dynamic nature of the **call** stack and the fact that each object needs its own private stack make coroutines harder to understand and less efficient to implement. The additional expressiveness they provide is not generally useful; programmers are not often faced with testing traversals of trees for equality.

CLU iterators are truly elegant. They are clear and expressive. They provide a single, uniform way to program all loops. They can be implemented efficiently on a single stack. Perhaps they have not caught on because, like general coroutines, they provide expressiveness in an arena where most programs do not need it. The only application I have ever found for which CLU iterators give me just what I need has been solving combinatorial puzzles,

and these don't crop up often.

Power loops have less to recommend them. They are not as clear as ordinary loops; if you don't believe this, consider what it means to **nest** a structure other than **for**. They don't provide any expressiveness beyond what recursion already provides. There are very few situations in which they are the natural way to pose an algorithm. However, the mathematical concept of raising a function to a power is valuable, and APL (discussed in Chapter 9), in which manipulation by and of functions is central, has an operator much like the power loop.

Io continuations provide a lot of food for thought. They spring from an attempt to gain utter simplicity in a programming language. They seem to be quite expressive, but they suffer from a lack of clarity. No matter how often I have stared at the examples of Io programming, I have always had to resort to traces to figure out what is happening. I think they are just too obscure to ever be valuable.

EXERCISES

Review Exercises

2.1 In what way is raising an exception like a **goto**? It what way is it different?

2.2 Write a CLU iterator `upto(a,b)` that yields all the integer values between `a` and `b`. You may use a **while** loop, but not a **for** loop, in your implementation.

2.3 Write a CLU iterator that generates all Fibonacci numbers, that is, the sequence 1, 1, 2, 3, 5, 8, . . . , where each number is the sum of the previous two numbers.

2.4 Write a Simula class `Fibonacci` with a field `Value` that the initialization code sets to 1 and then suspends. Every time the object is resumed, `Value` should be set to the next value in the Fibonacci sequence.

2.5 What does the following Io program do?

Figure 2.32

```
declare foo: → Number Continuation;                    1
     + Number 1 → More;                                2
     write More;                                       3
     Continuation .                                    4

foo 7;                                                 5
foo 9;                                                 6
terminate                                              7
```

2.6 Use power loops to initialize a $10 \times 10 \times 10$ integer array `A` to zeroes.

2.7 If I have power loops, do I need **for** loops?

Challenge Exercises

2.8 In Figure 2.3 (page 29), I show how a handler can reraise the same exception (or raise a different one) in order to propagate the raised exception further. Would it make sense to define a language in which exceptions were handled by the handler that raised them, not propagated further?

2.9 What are the ramifications of letting exceptions be first-class values? (First-class values are discussed in Chapter 3.)

2.10 Prove the contention on page 35 that when a CLU iterator terminates, indicating to its parent **for** loop that there are no more values, the iterator's activation record is actually at the top of the stack.

2.11 Use CLU iterators to write a program that takes a binary tree and prints all distinct combinations of four leaves.

2.12 Prove that in Figure 2.11 (page 36) the references to `Answer` generated in line 18 are always valid. In particular, prove that by the time an instance of `Answer` is deallocated, there are no remaining pointers to that instance. Actually, CLU requires that control variables, such as `T` in line 25, have a scope that only includes the loop they control. You may make use of this restriction in your proof.

2.13 Show how to use the C iterator macros to write a program that enumerates binary trees.

2.14 Are CLU iterators as powerful as Simula coroutines? In particular, can the binary-tree equality puzzle be solved in CLU?

2.15 In Figure 2.17 (page 44), could I replace the 9 in line 4 with the identifier `Number`?

2.16 What sort of runtime storage organization is appropriate for Io?

2.17 Does Io support recursion?

2.18 Show the execution trace of Figure 2.27 (page 48).

2.19 What is the meaning of a power loop for which the range is empty?

2.20 Figure 2.31 (page 51) has one potential inefficiency. What is it?

2.21 Power loops are modeled on **for** loops. Can I model them on **while** loops instead? That is, can they look like the following?

Figure 2.33

```
nest Boolean expression                                    1
     replicated part                                       2
do                                                         3
     body                                                  4
end;                                                       5
```

2.22 Does it make sense to place declarations inside the replicated part or the body of a power loop?