CSE305 Week 9: Control Structures (Sebesta Ch. 8)

Mashup of my old notes and Sebesta's new ones again:

Control Structures

Control Structures define the order of execution of program statements. E.g.,

- sequencing of statements (we take this for granted)
- if-then-else (and switch and match-case)
- loops

After opening with a similar outline that also mentions "unconditional branching" (i.e., goto) and "guarded commands" (which we've encountered via "when" in OCaml match-case), Sebesta gives a nice big-picture observation that there are "Levels of Control Flow"---

- Within expressions (Chapter 7)
- Among program units (Chapter 9)
- Among program statements (this chapter)

We are tracing flow within expressions and simple sequences of statements right now. On that score, here are some simple translations, supposing the declarations `int x,y,z,*p,*q;`

- `x = 3;` is `x 3 store pop`
- `y = x;` is `y x fetch store pop`
- `p = &x;` is `p x store pop`
- `q = p;` is `q p fetch store pop`
- `*q = 4;` is `q fetch 4 store pop`
- `y = *q;` is `y q fetch fetch store pop`
- `arr[i]` (Lvalue) is `arr e i fetch * +` (where `arr` is 0-based and `e` is
- `arr[i]` (Rvalue) is `arr e i fetch * + fetch` the element size in bytes)
- `x++` is `x fetch x x fetch 1 + store pop`
- `p++` is `p fetch p p fetch e + store pop` (`e` = size pointed at)
- `++x` is `(?)`

Note that the idea of these control structures is not dependent on any one programming language. Each of them is present in some form in all of the languages we've considered. Nor is the idea of stack-based evaluation---it's just more fun and hairy in C/C++. (Don't be misled by syntax: C has if-then-else structures, even though it omits the keyword "then". More subtly, C's "for" loop is *really* a while loop.)

Other Control Structures

case selection (`switch` in C/C++/Java)
for-loops
Procedure/Function Calls --> Chapter 9
goto (label); conditional go-to

(We will regard “break” or “exit” as part of other control structures, not ones in their own right.)

Sequencing works this way: After any non-branch instruction is executed, the **Instruction Counter** (IC) is automatically incremented to the next machine word, and so the next instruction is processed. But if a branch is taken, then the branch label value (b1) is copied into the IC -- and this causes execution to jump to the instruction held in memory location b1. The SPARC Book by R. Paul has eighteen (!) different conditional and unconditional branch instructions for integer tests alone (and more for floats). But JMPif0 is enough: it combines aspects of “if-then-else” and “while”. In fact,

Sequencing and JMPif0 form a complete set of assembly-level control structures.

Definition:
A set of control structures S is **complete** if every program P, in any language, can be translated into an equivalent (and similarly efficient) program P’ that uses only control structures in S.

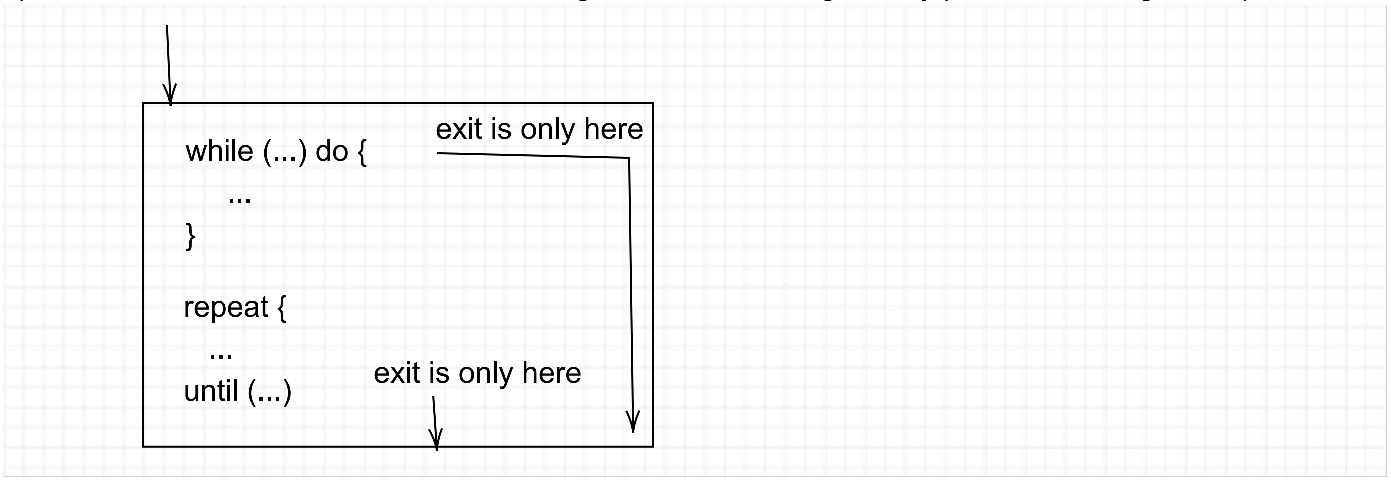
It is a famous fact that sequencing, if-then-else, and “while” form a complete set. These three structures (aka “Dijkstra’s triad”) suffice in any high-level language. Dijkstra used this fact as part of his argument that “goto” should be abandoned.
There is still some arument that for certain coding situations and on certain compilers using goto is more efficient than the alternatives. But 30+ years experience with goto-less structured programming has worked quite well, even in final-production code. Java has no goto statement.

The “One-In, One-Out” Principle

In BASIC or assembler, it is easy to write “Multi-In, Multi-Out” code like this:

```
.80  GOTO 140
...
100
110
120  IF ... THEN 300
...  ...
140  ...    (can be entered from 80, 130, or 300!)
...
200  IF ... THEN 400
...  ...
...
250  GOTO 500
...
300  IF ... THEN 140
...
```

For readability and correctness, it is important to minimize and localize the ways that control can spread. Modern control structures are designed to have a single entry point and a single exit point:



This is one reason why some languages discourage use of `break`. In Scala, `break` would interfere with the idea that loops yield a value at the end. OCaml has only "loop expressions" with no break allowed.

Procedure calls can complicate the picture, but the idea holds.
The “proper way” to code in BASIC used to be the “Double-Braided Spaghetti”:

```
100 IF <cond> THEN 200
110 {“ELSE” portion begins here ...
 :
 :
190 ... and ends here} GOTO 300
200 {“THEN” portion begins here...
 :
 :
290 ...up to here.  Done}
300 IF <cond> THEN 400
310 {“ELSE” portion ...
   :
   :
390 ... }  GOTO 500
400 {“THEN” ...
   :
   :
490 ... done}
500 IF... <etc>
```

Similarly with GOSUBS etc. If done well, this type of structure can be readily understood, but in practice it was abused, with long stretches of THEN and ELSE blocks, and errors in editing statement numbers, and the temptation to embellish this “Two-In Two-Out structure with more ins and outs... Now, even BASIC has adopted the familiar structured control units of FORTRAN, ALGOL-60, COBOL, Pascal, etc.

Comparing Conditionals

Ada has the most general form of “if-then-else”:

```
if <cond> then <stmt_seq>
{elsif <cond> then <stmt_seq>}
[else <stmt_seq>]
end if;
```

C and Pascal (1) lack the elsif option, (2) lack a closing “endif”, and (3) classify each branch as a statement rather than a sequence of statements. Of course, both do allow compound statements—via curly braces {...} in C and begin...end blocks in Pascal—so diference (3) isn’t very large (just a matter of clutter). How important are the other two?

(1) Lack of elsif forces one to use nested ifs

```
Ada                                         Pascal
if    age < 5  then ...;             if age < 5 then ...
elsif age < 13 then ...;                else
elsif age < 18 then ...;                   if age<13 then ...
else ...;                                     else
end if;                                          if age<18 then...
                                                    else ...;
```

In Ada, semicolon is a separator, if semicolon is a terminator then it ends all three individual IF statements!---similar in OCaml with `let`.

Here, Pascal’s nested ifs obscure the parallel structure that the “elsif” makes clear in Ada.

ML/OCaml’s “if-then-else” expression uses syntax similar to Pascal, except that the “else” is mandatory. This is similar to C’s conditional expression:

C: <cond> ? <exp1> : <exp2>
ML: if <cond> then <exp1> else <exp2>

However, ML and OCaml require that <exp1> and <exp2> have the same type. This extends to nested IFs (which happen when <exp1> or <exp2> is itself a conditional expression).

(2) The lack of a terminating “end if” can lead to the “Dangling Else” problem, as already discussed. Kernighan & Ritchie’s ANSI C book gives an example:

```
if (n >= 0)
  for (i= 0; i < n; i++)
    if (s[i] > 0
    {  print(“i = %d \n”, i);
       return i;
    }
else
  print (“n is negative”);   <-- WRONG
```

In the BNF, it is ambiguous whether the else belongs to the inner or outer IF. C and Pascal use the rule that such an else associates to the innermost IF for which it is legal. But this can produce problem like the above. Ada’s required “**`endif`**” terminator avoids the problem.

The CASE case

Ada is also considered to have the quintessential CASE statement:

```
case EXP is
   when CHOICES => STMT_SEQ
   {when CHOICES => STMT_SEQ}
end case;
```

where

```
CHOICES::=  CHOICE { ‘‘|’’ CHOICE}
CHOICE ::=  SIMPLE_EXP | DISCRETE_RANGE | others
```

Example:

```
case Paper_Rating(Paper) is
  when 0 .. 1 | 10 => Query(Referee);
  when 2 .. 3      => Reject(Paper);
  when 7 .. 9      => Accept(Paper);
  when 4 .. 5      => Print(‘‘Doubtful’’);
                      Examine(Paper);
  when 6           => Examine(Paper);
  when others      => raise Report_Error;
end case;
```

Notable features of the Ada case statement:

- EXP can have any discrete, countable type. (In C/C++/Java, EXP must be convertible to `int`.)
- The explicit keyword **`when`** labels each case alternative. Alternatives may be in any order, and need not be "consecutive.".
- More than one choice may be put on a case alternative line. A choice can be a range.
- Each statement sequence includes an automatic "break". Control does *not* "fall through" to other statements. (Each "**`when`**" after the first plays the role of "`break`" in the C `switch` statement.)
- Optional "**`others`**" allows a "default" branch.
- More than one stmt can be in a stmt seq.
- Explicit scope delimiter "end case;"
- The possible choices should be mutually exclusive, but need not be exhaustive. The compiler must "bomb" if any two choices overlap.
- The "case constants" and ranges must be fixed and known at compile time! This is not only needed for the last point, but also enables the compiler to build an efficient "jump table" from values to the corresponding stmt seqs. (This is also so in C/C++ and Java and Pascal.)

Standard Pascal lacks the vital convenience of a "default" or "others" clause for the case. Standard Pascal compilers would demand that all possible values of the case expression be explicitly included in the labeling of the alternatives. Many Pascal compiler vendors added an "others" or "otherwise" clause as a non-standard extension—and Ada standardized it.

Pascal and Ada both allow EXP and the case labels to belong to any "countable" type, meaning enumerated types (including. char) or integers or subranges thereof. C requires each labels to be an int, because it really is a label! Ranges and '|' groupings are not allowed in C, and the word "case" must appear before each individual label.

Switch in C/C++/Java

The reason for this and C's "fall-through" property in switch is the history of C's switch as a generalized goto statement! A C compiler standardly interprets "switch(exp)" as meaning "evaluate exp to get an integer v and then goto the the label case v:" To see this structure, let us not indent the statements corresponding to each case alternative, but rather "outdent" the case labels:

```
      switch (Paper_Rating(Paper)) {
case 0: case 1: case10: Query(Referee);break;
        case 2: case 3: Reject(Paper); break;
case 7: case 8: case 9: Accept(Paper); break;
        case 4: case 5: Print("Doubtful");
                        Examine(Paper);break;
                case 6: Examine(Paper);break;
               default: throw Report_Error;
       }
```

It is legal in C for any statement to have more than one label, so the body of the switch is just a sequence of labeled stmts. The body of the "`goto case v:`" causes execution to begin at the statement with label "`case v:`" and proceed from there. That is why we need to insert the "`break;`" (which the C compiler always treats as "go to the next enclosing curly brace `}`") in order to get the Ada/Pascal behavior.

C, Pascal and Ada all require that the identity of the "case constants" be determined and fixed at compile time. Therefore, you cannot do things like:

```
case Salary is
   when base => ...
```

when "`base`" is a variable. If "`base`" is a constant, so that Ada knows the value at compile time, and knows that it can't change during execution, then OK. By contrast, although ANSI C has a constant construct, e.g.

```
const int base = 30000;
```

the switch statement cannot trust it, because C does not mandate that an assignment to a constant, such as `base += 10000;` causes a compilation error. (Our ANSI C compiler in 2007 gave a warning.) Formalizing a capability where constants can be trusted at compile time is the essence of the recent development of the **`constexpr`** designator in C++.

Ada and Pascal compilers do in fact check at compile time that the case constants cover all the values in the discrete type of the case expression (with the help of the optional "others" for Ada), and that no duplicate values occur. C does the latter check, but omits the former—because it does not allow ranges and the type "`unsigned int`" is pretty big!

Here you can see some evolution toward the case-match in Scala and ML/OCaml. They have a "case expression" that can match on other types besides integers—indeed, it can match on any recursively defined datatype, including the structural patterns of lists, trees, and more.

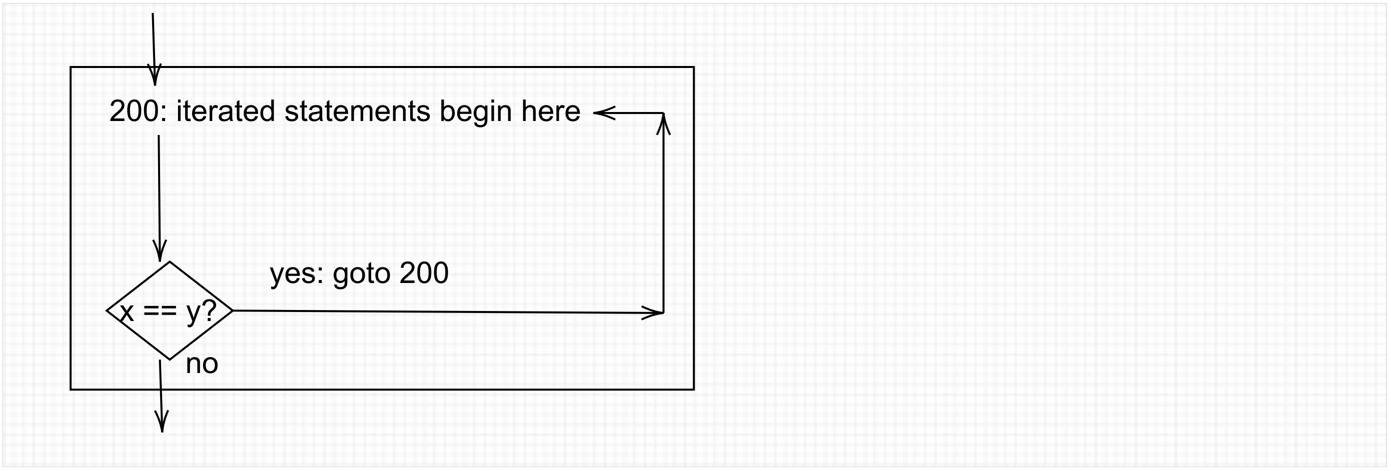
Lisp has a `cond` construct that allows arbitrary Boolean expressions before the "=>" (which is a space in Lisp). The first expression in sequence that is true determines the branch taken. This is really a kind of if-then-else construct with "guarded statements."

Loops: from low level to high level

Any program that is not "straight line" must have a loop or other backward jump in the control somewhere. The backward jump can come from:

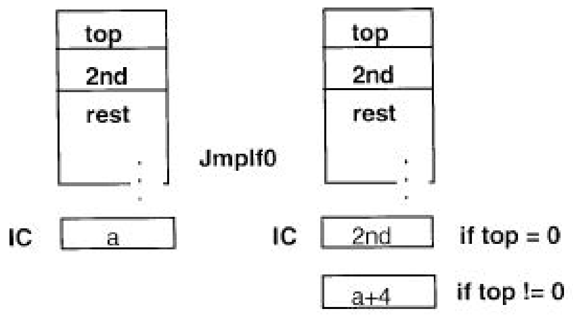
- a GOTO statement that jumps backward;
- a structured loop stmt, in which control, upon hitting the bottom-of-loop, automaticaly goes back to the top; or
- recursive procedure calls.

Ignoring recursion for now, you can build a loop with a test and GOTO:



This obeys the "One-In, One-Out" principle and was always considered a reasonable use of GOTO. Note too that the test x == y could have been written as x - y == 0. More generally, if you convert true = 1 and false = 0, then any condition could be tested as (1 - cond) == 0.

In fact, the Java Virtual Machine (which is expressly stack-based) has an if_cmp_cond instruction like the above, and has some other jump instructions---as do other compilation tagrets. But we can imagine using just a jmpif0 instruction. All it needs is to add an Instruction Counter (IC) to our rudimentary stack language. Here is a simplified picture for a 32-bit machine where **sequencing** means advancing the IC address **a** by 4 bytes.



If we want to clear the value compared and the address 2nd from the stack after the test, just do **`jmpif0 pop pop`**.

Given that "Dijkstra's Triad" of sequencing, if-then-else, and a simple test-loop are enough for universal programming, and given that we are going to compile into a simple jump construct anyway, why bother with extra control structures?

> The real purpose of control structures is to reflect the natural ways programmers think about solving problems, and to channel their thoughts into code blocks that are easy to read and maintain. [Steve McConnell, *Code Complete: A Practical Handbook of Software Construction*.]

A secondary reason is that greater control flexibility can sometimes help one produce more efficient object code. This influenced choices made in C, which grew out of the PDP-11 instruction set. Pascal opted instead for a simple compiler with a restricted set of control structures. ALGOL-60 had more flexibility than Ada kept and that anyone has ever really used.

Issues for Loops

Sebesta's agenda, in a slide titled "Iterative Statements":

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- General design issues for iteration control statements:
    1. How is iteration controlled?
    2. Where is the control mechanism in the loop?

Sebesta goes on to consider counter-controlled versus logic-controlled loops, but I want to bring that and four other issues to top level:

1. Is the loop such that from its header alone---regardless of the statements in its body---the runtime system can, upon entering the header, place a bound on the maximum number of iterations possible? If so, call it an "abstract for-loop", else it is "properly a while-loop." If the exact number of iterations is known, call it "strictly counted."
2. Whether for-loop or while-loop, is breakout apart from the header or the main conditional test allowed?
3. Is the iteration scheme of the loop itself an object in the program?
4. Does the loop `yield` a value or values---at the end and/or along the way?

The main drift of PL design regarding iteration, IMPHO, is that the only saintly way to get a strictly-counted for-loop is to not have a counter.

The MAD C for loop

The for-loop in C originated around 1964 in a language called “MAD” for “Multiple Algorithmic Decoder.” Observe that in its abstract form,

```
    for (E1; E2; E3) <stmt>
```

it is equivalent to the C while loop

```
E1;
while (E2) {
  <stmt>
  E3;
}
```

except that `continue` jumps to E3 in the `for` loop, but jumps past E3 to "**}**-and-repeat" in the `while` loop. The equivalent code in Ada (which is taken as representative of languages of the 1960s to 1980s that used keywords rather than braces to delimit blocks) is:

```
E1;
while E2 loop
  <stmt_seq>
  E3;
end loop;
```

Note that E1 and E3 are "really" assignment statements, but C/C++ officially classes assignment statements as expressions. In Pascal and Ada they are classed as statements. C and C++ allow general use of the "comma operator" to sequence expressions, but Java restricts its use to E1 inside for loops. Going the other way, the while-loop

```
while (cond) { stmts }
```

is equivalent to the "for-loop"

```
for ( ; cond; ) { stmts }
```

In both cases, the stmts need to update something in `cond` so that it could become false. Here is a concrete example, searching for (the first occurrence of) an element x in an array A:

```
for (i = 0; x != A[i]; i++) ;
```

Yes, an empty loop body! The following while-loops in C and Ada are equivalent to this:

```
i = 0;
while (x != A[i]) i++;

i := 0;
while x /= A(i) loop
   i := i + 1;
end loop;
```

Note that if x does not appear in A, then all of these three code segments will cause i to go beyond the bounds of the array. In C and C++ the code will merrily (and unsafely!) keep fetching from consecutive memory positions. Modern languages require throwing an exception for any overshoot of array bounts. A standard programming trick is to make the “sentinel assignment” `A[n-1] = x;` before entering the loop, so that `i == n-1` after the loop equates to failure. Then C and C++ reap benefits in speed from not having to slow down for safety.

Bounded Iteration

Thus far we’ve discussed C’s for loop and how it relates to while and recursion. However, programmers in other languages usually think of for-loops only as bounded (or “counted”) iteration controls. Here's hwo they look in various languages besides C/C++/Java:

```
COBOL:    PERFORM <grouped statements>
          VARYING J FROM 1 BY 1 UNTIL J > N

BASIC:    FOR J=1 TO N
            <statements>
          NEXT J;

Fortran:  DO 10 J = 1,N
             <statements>
          10  CONTINUE

Pascal:   for J:= 1 TO N do
          begin
             <statements>
          end

Ada:      for J in 1..N loop
             <statements>
          end loop;

Python:   for j in range(1, N+1):
             <indented-statements>

Scala:    for ( j <- 1 to N ) {
             <statements>
          }
```

Here are features common to all of these languages:

- Single loop variable (here “J”), generally of integer type.
- Loop increment is +1 by default.
- Loop bounds may be variables, so long as their values are known at initial loop entry time so that the number of iterations is known in advance.
- Some type of end-of-loop delimiter is used---except in Python the delimiter is stopping the indentation.

One topical difference is whether N should be inclusive or exclusive.   You can get exclusivity in Scala by saying "`until`" rather than "`to`". The following two issues involving the loop variable are deeper in the sense of being more bug-prone:

- Should J be visible outside the scope of the loop?
- May J and/or loop bounds such as N be re-assigned during the loop? e.g.

Besides C, other older languages allowed doing things like this:

```
  for J = 1 to N do
    ...
    if a(J) < 0 then J = J+3; /*skip 3 places*/
    else ...
  end;
```

The Ada for loop bridged "old" and "new" attitudes:

```
[LABEL:]  for ID in [reverse] DISCR_RANGE loop
      STMT_SEQ
      end loop [LABEL]; --labels if there must match
```

Here `DISCR_RANGE` can be a subrange of the integers such as 1..N, or it can be (a subrange of) an enumerated type (e.g.: “for Day in Week loop...”). The increment goes to the next-greater element of the discrete range, except that optional reverse gives a step to the next-lesser. The optional loop label is used for possibly breaking out of multiply-nested loops, as treated below.

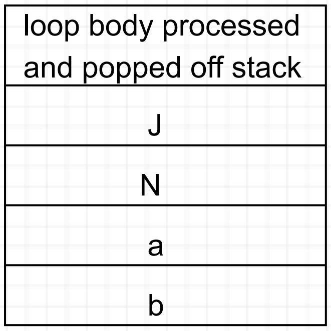
To answer the two questions for Ada:

- No, the loop variable is implicitly declared in the loop header and is invisible outside the loop. It is treated the same as a constant parameter to a subprogram, so cannot be altered in the loop body.
- No, the range is read and fixed when the loop is entered and cannot be affected.

These strictures enable compiler optimization of loops. We can get a hint of this with our "rudimentary stack language.: To compile

```
for J in 1..N loop
  <statements>  -- beginning at memory address a
end loop;
<next stmt>     -- beginning at memory address b
```

first push a, then push N, then push J. Then translate the body of the loop. It will be processed above J on the stack. Finally translate "end loop" with a JMPif0 instruction, *modified* as "`jmpifequal`", so that if N ≠ J, a is copied into the IC (so the loop repeats), while if J = N, all three items are popped and control naturally "falls through" to b.



The point is that N (and often J too) need not be re-fetched in order to be tested.

ANSI C drew a page from Ada by allowing the loop variable to be declared inside the loop:

```
for (int J = 1; J <= N; J++) {
   <statements>
}
```

Then, however, the variable J is not visible outside the loop body. Java adopted this idea. Theoretically this is supposed to work even if a different variable "J" is in scope just outside the loop, with that J being left unaffected, but the ANSI C and C++ compilers we have seem to hiccup in this case! Overall, there seems to be no firm consensus on the question of J's scope. However, there is a fairly strong consensus on the second question, namely that allowing J or N to be altered inside the loop is a bad idea, for both users and compilers! PL/1 was especially bedeviled by this.

Breaking Out

Ada allows "multiple breakout" via named loops.

```
. . .
OUTER: for i in 1..m loop
  INNER: for j in 1..n loop
    if A(i, j) = Sought_Value
      then exit OUTER;
    end if;
  end loop INNER;
end loop OUTER;
```

There have been arguments about whether “multiple breakout” is too powerful a jump (like `goto`) and whether it was hard on compilers. The fact that Ada embraces it indicates that people consider it safe and readable enough. It is certainly neater than cluttering up nested loops with extra Boolean flags!

Ada offers two other devices that help compilers minimize the overhead of executing loops. The “short-cricuit evaluation” of `and then` (which equals the standard behavior of && in C/C++/Java), and exceptions, which will be treated later. Here is an example of the former.

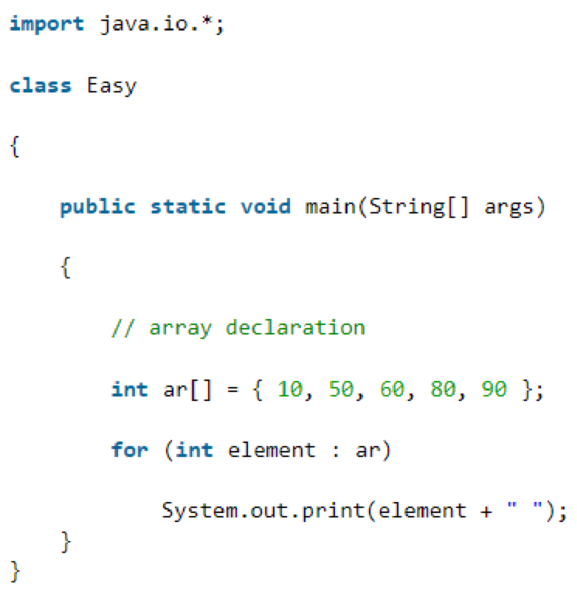
```
function Sumarray(N: in integer; A in IntArray;
              Sum out natural) is
  -- A has bounds 1..N.  Function sums contents of A
  -- so long as they are positive values.
  k: natural := 1;
begin
  sum := 0;
  while k <= N and then A(k) > 0 loop
    sum := sum + A(k);
    k := k + 1;
  end loop;
end Sumarray;
```

The short-circuit evaluation of and then allows us to avoid run-time errors without extraneous flags, but it’s not necessarily the best way to write the loop. A “breakout” could make it more explicit that the loop is being aborted early because of a non-positive value.

```
function Sumarray(A in IntArray; Sum out natural)  is  --Sum contents of A so long as
they are positive.
  Neg-Entry: exception;
begin
  sum := 0;
  for k in A’range loop
    if A(k) <= 0 then raise Neg-Entry; end if;
    sum := sum + A(k);
  end loop;
exception
  when Neg-Entry =>
    put_line(‘‘Negative Entry found’’);
end Sumarray;
```

Counted Loops Without Counting

The one (almost-)sure way to indicate that you want a strictly counted for-loop is to iterate over the *data*, not with a *counter*. This feature has been added to languages that didn't originally have it, e.g. Java:



```java
import java.io.*;

class Easy

{

    public static void main(String[] args)

    {

        // array declaration

        int ar[] = { 10, 50, 60, 80, 90 };

        for (int element : ar)

            System.out.print(element + " ");
    }
}
```

Some languages distinguish this via the "`foreach`" keyword. C++ has a whole generalized interation scheme (instead).

Recursion as a Loop Structure

The basics of an unbounded loop are 1. initialization, 2. tests, and 3. incrementing. Here is how these three are reflected by recursive function calls:

1. Initialize via the parameters in the initial call.
2. Test in the function body to determine when (or whether) to stop the recursion.
3. Increment by changing the values passed to the next recursive call.

A simple example in C++ (using “#define null 0”):

```cpp
int sumlist(Intlist* curr)
{  if (curr == null) return 0;
   else return curr->val +
                 sumlist(curr->next);
}
```

The increment is from `curr` to `curr->next`. If we assume that the “`Intlist`” object is a linked list whose last item has a null next pointer, then we have a *sentinel*.

Recursion is considered inefficient in C/C++ because compilers generally allocate a stack frame for each recursive call, at considerable time overhead. In ML and OCaml, however, the form of the language is so regularized that compilers are often able to convert recursions into iterations that are at least as efficient as the equivalent C code.

[My previous Ch. 8 notes ended with Tail Recusion which has only a brief mention in the text and also goes with chapter 9 anyway, so I will pick it up there.]