Chapter 8

Statement-Level
Control Structures

Chapter 8 Topics

- Introduction
- Selection Statements
- Iterative Statements
- Unconditional Branching
- Guarded Commands
- Conclusions

Levels of Control Flow

- Within expressions
  - Controlled by operator precedence and associativity
- Among program units
  - Function call and concurrency
- Among program statements
  - Control statements and control structures

Evolution of Control Statements

- FORTRAN I control statements were based directly on IBM 704 hardware
- Much research and argument in the 1960s about the issue
  - One important result: It was proven that all algorithms represented by flowcharts can be coded with only two-way selection and pretest logical loops (Bohm and Jacopini, Structured Programming Theorem, 1966)
  - Any language with these features is "Turing-complete" – can compute anything that is computable

Goto Statement

- At the machine level we really have only conditional and unconditional branches (or gotos)
- Using conditional gotos we can create any kind of selection or iteration structure
- But undisciplined use of gotos can create spaghetti code

Control Structures

- A control structure is a control statement and the statements whose execution it controls
- Programmers care more about readability and writ ability than theoretical results
  - While we can build very small languages and/or use very simple control structures, we would like to an expressive language with readable code
  - Languages often provide multiple control structure for iteration and selection to aid readability and writability

Design Issues

- Multiple Exits
  - Not really a design issue at all.
  - All languages allow for multiple exits
    - If the target is always the first statement after the end of the structure, not an issue
    - If the target is unrestricted then the question reduces to whether or not we allow GOTO statements
- Multiple Entry Points
  - Only possible if language already includes goto statements and labels (names for statements and thus possible targets)

Selection Statements

- A selection statement provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Selection Statement

- IfStatement → if (Expression) Statement
  [else Statement]
- Example:

```
if (a > b)
  z = a;
else
  z = b;
```

- If the test expression is true, then the output state of the conditional is the output state of the then branch,
- else the output state of the conditional is the output state of the else branch.

Design Issues

- What is the form and type of the control expression?
- How are the **`then`** and **`else`** clauses specified?
- How should the meaning of nested selectors be specified?

Control Expression

- If the **`then`** reserved word or some other syntactic marker is not used to introduce the then clause, the control expression is placed in parentheses (typical of C-like languages)
- Early C did not have a Boolean
  - Selection clauses were integer or arithmetic expressions
  - Many languages transparently coerce the result of a control expression to Boolean
    - 0 = false, non-zero = true
    - Empty string = false, non-empty string = true
    - But watch out: some languages coerce the string to an integer first, then test for 0
- In languages such as Ada, Java, Ruby, and C#, the control expression must be explicitly Boolean

Clause Form

- In many contemporary languages, the then and else clauses can be single statements or compound statements
- In Perl, all clauses must be delimited by braces (they must be compound)
- In Fortran 95, Ada, and Ruby, clauses are statement sequences
- Python uses indentation to define clauses

```
if x > y :
  x = y
  print "case 1"
```

- In Python : or then can be used

Nesting Selectors

- Java example

```
if (sum == 0)
   if (count == 0)
       result = 0;
else result = 1;
```

- Which `if` gets the `else`?
- Java's static semantics rule: `else` matches with the nearest `if`

Nesting Selectors (continued)

- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
  if (count == 0)
      result = 0;
}
else result = 1;
```

- The above solution is used in C, C++, and C#
- Perl requires that all then and else clauses to be compound

Ending with Reserved Words

- Can avoid the issue of nested selection statements using a reserved word to end clauses
- Ex: Ruby

```
if sum == 0 then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Nesting Selectors (continued)

- Python

```
if sum == 0 :
  if count == 0 :
    result = 0
  else :
    result = 1
```

Multiple-Way Selection Statements

- Select among any number of control paths
- We really only need one way to express selection semantics
  - We can use a multi-way selector to express 2-way selection semantics
  - We can use a 2-way selector to express multi-way selection semantics
- Either alternative is syntactically clumsy and makes programs harder to read and write.

Two types of multiway selection

- Multiway selection is used for two different, but similar purposes:
  1. Providing multiple control paths based on the value of a single scalar with a relative small range of possible ordinal values
  2. Flattening deeply nested if statements consisting of mutually-exclusive cases
- Case or Switch statements are usually used for the first purpose and else-if statements for the latter
  - Some languages combine both purposes into a single flexible case statement

Case/Switch Design Issues

1. What is the form and type of the control expression?
2. How are the selectable segments specified?
3. Is execution flow through the structure restricted to include just a single selectable segment?
4. How are case values specified?
5. What is done about unrepresented control expression values?

Switch or Case statement

- Selection of a small set of ordinal values
  - Started with Fortran computed GOTO
    `GO TO (100, 87, 345, 190, 52) COUNT`
  - Semantics: if count = 1 goto 100, if count = 2 goto 87 etc.
  - Can be implemented as a jump table
- Switch or Case entry statement contains a control expression
- Body of statement contains multiple tests for values of control expression with associated block of code
- For efficient implementation (jump table) control expression should resolve to relatively small number of discrete values

Switch in C-Like Languages

```
switch(n) {
  case 0:
    printf("You typed zero.\n");
    break;
  case 1:
  case 9:
    printf("n is a perfect square\n");
    break;
  case 2:
    printf("n is an even number\n");
  case 3:
  case 5:
  case 7:
    printf("n is a prime number\n");
    break;
  case 4:
    printf("n is a perfect square\n");
  case 6:
  case 8:
    printf("n is an even number\n");
    break;
  default:
    printf("Only single-digit numbers are allowed\n");
  break;
}
```

C Switch

- Design choices for C's **`switch`** statement
  1. Control expression can be only an integer type
  2. Selectable segments can be statement sequences, blocks, or compound statements
  3. Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)
  4. **`default`** clause is for unrepresented values (if there is no **`default`**, the whole statement does nothing)
- The C switch statement was designed to be as flexible as possible
- For nearly all normal usage the flexibility is much greater than needed and the requirement for an explicit break to terminate execution seems like a design error

C# Changes

- C# has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (**`goto`**, **`return`**, **`continue`** or **`break`**)
- Control expression and the case constants can be strings

C#

```
switch (expression)
{
   case constant-expression:
      statement
      jump-statement
   [default:
      statement
      jump-statement]
}
```

C#

```
switch (value){
  case -1:
     minusone++;
     break;
  case 0:
     zeros++;
     goto case 1;
  case 1:
     nonnegs++;
     break;
  default:
     return;
}
```

An interesting (ab)use of switch in PHP

```
function flavor($type = null) {
    switch ($type) {
        default:
            $type = null;
        case $array[] = "chocolate":
            if ($type != null) {
                $array = array($type);
                break; }
        case $array[] = "strawberry":
            if ($type != null) {
                $array = array($type);
                break;
            }
        case $array[] = "vanilla":
            if ($type != null) {
                $array = array($type);
                break;
            }
    }
    if ( (count($array) != 1) ) {
        return "Flavors available: " . implode(", ", $array);
    } else {
        return "Flavor selected: " . implode(", ", $array);
    }
}
```

An interesting (ab)use of switch in PHP

- Functionality is attributable to semantics of assignment expressions

```
echo flavor() . "<br>";
/* Flavors available: chocolate,
  strawberry, vanilla */
echo flavor("banana") . "<br>";
/* Flavors available: chocolate,
  strawberry, vanilla */
echo flavor("chocolate") . "<br>";
/* Flavor selected: chocolate */
```

Ada

- Ada

```
case expression is
    when choice list => stmt_sequence;
    …
    when choice list => stmt_sequence;
    when others => stmt_sequence;]
end case;
```

- More reliable than C's **`switch`** (once a stmt_sequence execution is completed, control is passed to the first statement after the **`case`** statement

1-28

Case in Ada

```
type Directions is (North, South, East, West);
   Heading : Directions;
   case Heading is
      when North =>
         Y := Y + 1;
      when South =>
         Y := Y - 1;
      when East =>
         X := X + 1;
      when West =>
         X := X - 1;
   end case;
```

Ada also supports choice lists:

```
case ch is
      when 'A'..'Z'|'a'..'z' =>
```

Ada Design Choices

- Ada design choices:
  1. Expression can be any ordinal type
  2. Segments can be single or compound
  3. Only one segment can be executed per execution of the construct
  4. Unrepresented values are not allowed
- Constant List Forms:
  1. A list of constants
  2. Can include:
     - Subranges
     - Boolean OR operators (|)

Switch in Ruby

```
case n
when 0 then       puts 'You typed zero'
when 1, 9 then    puts 'n is a perfect square'
when 2 then
  puts 'n is a prime number'
  puts 'n is an even number'
when 3, 5, 7 then puts 'n is a prime number'
when 4, 6, 8 then puts 'n is an even number'
else              puts 'Only single-digit numbers are allowed'
end
```

- Switch can also return a value in Ruby

```
catfood = case
          when cat.age <= 1: junior
          when cat.age > 10: senior
          else               normal
          end
```

Perl, Python, Lua

- Perl, Python and Lua do not have multiple-selection constructs but the same effect can be obtained using else-if structures

Implementing Multiple Selection

- Four main techniques
  1. Multiple conditional branches
  2. Jump tables (indexing into array)
  3. Hash table of segment labels
  4. Binary search table
- Quiz Nov 4
  - Draw diagrams illustrating options 2,3,4 where the segment labels and associated addresses are
    - 0, 2, 3 addr1
    - 1 addr2
    - 4,5,6 addr3
    - 7 addr4

A Deeply Nested If

```
if (grade > 89) {
  ltr = 'A';
} else {
  if (grade > 79) {
      ltr = 'B';
  } else {
      if (grade > 69) {
            ltr = 'C';
      } else {
            if (grade > 59) {
                  ltr = 'D';
            } else {
                  ltr = 'E';
            }
      }
  }
}
```

Else If

```
if (grade > 89) {
  ltr = 'A';
} else if (grade > 79) {
  ltr = 'B';
} else if (grade > 69) {
  ltr = 'C';
} else if (grade > 59) {
  ltr = 'D';
} else
  ltr = 'E';
}
```

Multiple-Way Selection Using `if`

- Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses, for example in Python:

```
if count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

Multiple-Way Selection Using **if**

- The Python example can be written as a Ruby case

```
case
  when count < 10 then bag1 = true
  when count < 100 then bag2 = true
  when count < 1000 then bag3 = true
end
```

Flexibility of If-Elseif

```
If (today = Weds AND I < 10) Then
  Do thing1
ElseIf (y * 17 == g(2) OR notlegal) Then
  Do thing2
 . . .
```

Operational Semantics of If-Elseif

```
If e1 goto 1
If e2 goto 2
If e3 goto 3
. . .
1: S1
   S2
  goto out
2: S1
   S2
  goto out
.  .  .
```

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- First iterative constructs were directly related to array processing
- General design issues for iteration control statements:
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

Loop Control

- The body is the collection of statements controlled by the loop
- Several varieties of loop control:
  - Test at beginning of loop (While)
  - Test at end of loop (Repeat)
  - Infinite (usually terminated by explicit jump)
  - Count-controlled (restricted While)
- Note that beginning and end are logical, not physical

Count-Controlled Loops

- A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values
  - Note that some machine architectures directly implement count controlled loops (e.g., Intel LOOP instruction)
- Design Issues:
  1. What are the type and scope of the loop variable?
  2. Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
  3. Should the loop parameters be evaluated only once, or once for every iteration?

Fortran 95 DO Loops

- FORTRAN 95 syntax

```
DO label var = start, finish [, stepsize]
```

- Stepsize can be any value but zero
- Parameters can be expressions
- Design choices:
  1. Loop variable must be **INTEGER**
  2. The loop variable cannot be changed in the loop, but the parameters can; because they are evaluated only once, it does not affect loop control
  3. Loop parameters are evaluated only once

Operational Semantics

```
Init_val = init_expression
Term_val = terminal_expression
Step_val = step_expression
Do_var = init_val
It_count = max(int(term_val – init_val +
  step_val)/step_val,0)
Loop:
     if it_count <= 0 goto done
     [body]
     do_var = do_var + step_val
     it_count = it_count + 1
     Goto loop:
Done:
```

Ada For Loop

- Ada

```
for var in [reverse] discrete_range loop
...
end loop
```

- Design choices:
  - Type of the loop variable is that of the discrete range (A discrete range is a sub-range of an integer or enumeration type).
  - Loop variable does not exist outside the loop
  - The loop variable cannot be changed in the loop, but the discrete range can; it does not affect loop control
  - The discrete range is evaluated just once
- Cannot branch into the loop body

Operational Semantics

```
[define for_var with type = discrete_range]
[evaluate discrete_range]
Loop:
     if [no more elts of discrete_range]
          goto done
     for_var – next element of discrete_range
     [loop body]
   Goto loop:
Done:
  [undefine for_var]
```

C-style Languages

- C-based languages

```
for ([expr_1] ; [expr_2] ; [expr_3])
   statement
```

  - All expressions are optional
  - Expressions can be multiple statements, separated by commas
  - Value of list of expressions is value of last expression

C-Style For Loops

- The general form:

```
for (expressions1;  expression2; expressions3)
  statement;
```

- Semantically equivalent to

```
expressions1;
while (expression2) {
  statement;
  expressions3;
}
```

Example

- The C-style for loop is really a while loop

```
sum = 0.0;
for (j = 0; j < SIZE; j++)
  sum += a[j];
OR:
for (sum =0.0, j = 0; j < SIZE; j++)
  sum += a[j];
```

- Semantically equivalent to:

```
sum = 0.0;
j = 0;
while (j < SIZE) {
sum+= a[j];
j++;
}
```

C-Style For Loops

- If the second (test) expression is absent is considered TRUE so a for loop without a second expression is potentially infinite
- Design choices:
  - No explicit loop variable
  - Anything and everything can be changed in the loop
  - Legal to branch into loop body
- Note C's flexible, unsafe, anything goes culture vs. Ada's prevent errors at the expense of flexibility

C-Style For Loops

- Useful C style loops can have empty bodies

```
for (count1 = 0, count2 = 1.0;
     count1 <= 10 && count2 <= 100.0;
     sum += ++count1 + count2, count2 *= 2.5)
  ;

for (sum =0.0, j = 0; j < SIZE; sum += a[j]++)
  ;
```

- This is because **`expressions3;`** is really part of the loop body
- An infinite Loop

```
for (;;)
  doSomething;
```

C to C++ and Java

- C++ differs from C in two ways:
  1. The control expression can also be Boolean
  2. The initial expression can include variable definitions (scope is from the definition to the end of the loop body)

     `For (int count=0; count < max; count++){ … }`
- Java and C#
  - Differs from C++ in that the control expression must be Boolean

Python

- Python

```
for loop_variable in object:
   - loop body
  [else:
   - else clause]
```

- The object is often a range, which is either a list of values in brackets ([2, 4, 6]), or a call to the range function (range(5), which returns 0, 1, 2, 3, 4
  - Range can take an optional lower bound: range(2,7) returns [2,3,4,5,6]
  - Range can take an optional step size: range [0,8,2] returns [0,2,4,6]
  - Range can only accept integer arguments
- The loop variable takes on the values specified in the given range, one for each iteration
- The else clause, which is optional, is executed if the loop terminates normally
  - It is not executed when a break statement terminates the loop

Logically-Controlled Loops

- Repetition control is based on a Boolean expression
- Much simpler than count-controlled loops
- Design issues:
  - Pretest (while) or posttest (repeat) ?
  - Allow arbitrary exits?
  - Should the logically controlled loop be a special case of the counting loop statement or a separate statement?

Pretest Loops

- **WhileStatement → while (Expression) Statement**
  - The expression is evaluated.
  - If it is true, first the statement is executed, and then the loop is executed again.
  - Otherwise the loop terminates.
  - The loop body is not guaranteed to execute at all.
- Adequate for all looping needs

Pretest Loops

- **WhileStatement → while (Expression) Statement**
  - The expression is evaluated.
  - If it is true, first the statement is executed, and then the loop is executed again.
  - Otherwise the loop terminates.
  - The loop body is not guaranteed to execute at all.
- Adequate for all looping needs

Pretest Loop Operational Semantics

```
loop:
 if (control_expression==false) goto out
 [loop  body]
 goto loop
out:
  . . .
```

Posttest or Repeat Loops

- Test at end of loop; body executes at least once

```
DoWhileStatement →
  do
   Statement
  while (Expression)
```

- Different keywords may change sense of test

```
do
  Statement
until(Expression)
```

Posttest Loop Operational Semantics

```
loop:
 [loop  body]
if (control_expression==true) goto loop
out:
. . .
```

- With "until"

```
loop:
 [loop  body]
until control_expression
  /* if (control_expr==false) goto loop *?
out:
 . . .
```

C While and Do

- C and C++ have both pretest (while) and posttest forms, in which the control expression can be arithmetic:

```
while (ctrl_expr)      do
  loop body              loop body
                       while (ctrl_expr)
```

- Java is like C and C++, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no **goto**

Ada Loops

- Ada like many languages allows arbitrary tests:

```
loop
  Get(Current_Character);
  exit when Current_Character = '*';
end loop;
```

  - The general Ada form allows both pretest (while) and posttest (repeat) loops
- Ada also has a while loop

```
while Bid(N).Price < Cut_Off.Price loop
   Record_Bid(Bid(N).Price);
   N := N + 1;
end loop;
```

Loop Control and Exit

- Sometimes it is convenient for the programmers to decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (e.g., `break`)
- Design issues for nested loops
  1. Should the conditional be part of the exit?
  2. Should control be transferable out of more than one loop?

Loop Control

- C provides two goto-like constructs
  - break (exit current loop / switch structure)
  - continue (transfer control to loop test)
- In C, C++, and Python **`continue`** is unlabeled; skip the remainder of the current iteration, but do not exit the loop
- Java and Perl have labeled versions of **`continue`**
- Ada has an exit statement similar to break

Data Structure Based Iteration

- Control mechanism is a call to an *iterator* function that returns the next element in some chosen order, if there is one; else loop is terminate
- An iterator is an object that has a state
  - It has to remember the last object returned

```
init_iterator(it);
while (obj = it.getNextObject()) {
   process_obj(obj);
}
```

(side note: this code is "phpish." Most iterators return an object if available and FALSE if not)

Data Structure Based Iteration

- C's **`for`** can be used to easy program a user-defined iterator:

```
for (p=root; p==NULL; p = p->next){
   process_node(p);
   . . .
}
```

Python For statement

- **The For statement is really an iterator**
- The for statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:
- **for_stmt** ::= "for" target_list "in" expression_list ":" suite ["else" ":" suite] The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the expression_list. The suite is then executed once for each item provided by the iterator, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the else clause, if present, is executed, and the loop terminates.
- A break statement executed in the first suite terminates the loop without executing the else clause's suite. A continue statement executed in the first suite skips the rest of the suite and continues with the next item, or with the else clause if there was no next item.

Python For Statement

- The suite may assign to the variable(s) in the target list; this does not affect the next item assigned to it.
- The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop. Hint: the built-in function range() returns a sequence of integers suitable to emulate the effect of Pascal's for i := a to b do; e.g., range(3) returns the list [0, 1, 2].
- Note
- There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,
- for x in a[:]: if x < 0: a.remove(x)

PHP foreach

- PHP4 introduced a foreach iterator for arrays

```
foreach (array_expression as $value)
   statement
foreach (array_expression as $key => $value)
   statement
```

- PHP 5 introduced iteration over objects

Iterating over PHP object properties

```
<?php
   class MyClass
   {
        public $var1 = 'value 1';
        public $var2 = 'value 2';
        public $var3 = 'value 3';
        protected $protected = 'protected var';
        private   $private   = 'private var';

        function iterateVisible() {
           echo "MyClass::iterateVisible:\n";
           foreach($this as $key => $value) {
               print "$key => $value\n";
           }
        }
   }

   $class = new MyClass();
   foreach($class as $key => $value) {
        print "$key => $value\n";
   }
   echo "\n";
   $class->iterateVisible();
?>
```

Output of Example

```
var1 => value 1
var2 => value 2
var3 => value 3
MyClass::iterateVisible:
var1 => value 1
var2 => value 2
var3 => value 3
protected => protected var
private => private var
```

Iterators

PHP explicit iterator

- `current` points at one element of the array
- `next` moves `current` to the next element
- `reset` moves `current` to the first element

- Java
  - For any collection that implements the `Iterator` interface
  - `next` moves the pointer into the collection
  - `hasNext` is a predicate
  - `remove` deletes an element
- Perl
  - has a built-in **`foreach`** iterator for arrays and hashes

Iterators

- Java 5.0 uses foreach)
  - For arrays and any other class that implements the Interable interface

```
for (String myElement : myList) { … }
```

- C#'s foreach statement iterates on the elements of arrays and other collections:

```
Strings[] = strList = {"Bob", "Carol", "Ted"};
foreach (Strings name in strList)
    Console.WriteLine ("Name: {0}", name);
```

  - The notation `{0}` indicates the position in the string to be displayed

Unconditional Branching

- Transfers execution control to a specified place in the program
- Represented one of the most heated debates in 1960's and 1970's
  - Major concern: Readability
  - Some languages do not support **goto** statement (e.g., Java)
  - C# offers **goto** statement (can be used in **switch** statements)
- Loop exit statements are restricted and somewhat camouflaged `goto`'s

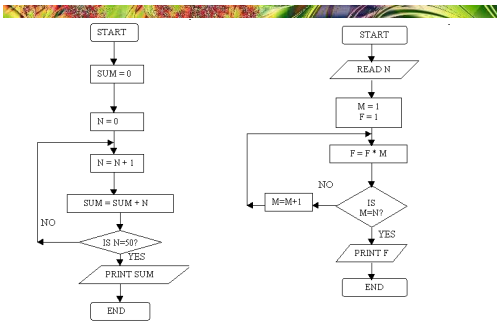
The Goto Controversy

- In 1968 Edsger Dijkstra wrote a letter to the editor of *Communications of the ACM* entitled "GoTo Considered Harmful."
- In the 1960's the major program design tool was the flowchart
- Fortran and Basic were written with line numbers
- Programs naturally resembled flowcharts

Structured Programming

- Dijkstra advocated eliminating the use of the GOTO statement in favor of conditional and iterative structures
- Development of C and Pascal with the requisite control structures started the "structured programming revolution."
- Both languages still have goto statements, but they are rarely used.

Flowchart Examples



Fortran 66 Spaghetti Code

```
SUBROUTINE OBACT(TODO)
      INTEGER TODO,DONE,IP,BASE
      COMMON /EG1/N,L,DONE
      PARAMETER (BASE=10)
   13 IF(TODO.EQ.0) GO TO 12
      I=MOD(TODO,BASE)
      TODO=TODO/BASE
      GO TO(62,42,43,62,404,45,62,62,62),I
      GO TO 13
   42 CALL COPY
      GO TO 127
   43 CALL MOVE
      GO TO 144
  404 N=-N
   44 CALL DELETE
      GO TO 127
   45 CALL PRINT
      GO TO 144
   62 CALL BADACT(I)
      GO TO 12
  127 L=L+N
  144 DONE=DONE+1
      CALL RESYNC
      GO TO 13
   12 RETURN
      END
```

A good use of gotos

- Natural implementation of DFSA

```
State0:
  ch = getchar();
  if (ch =='0')
    goto State1;
  else
    goto State2;
State1:
  while ((ch = getchar()) == '0')
    ;
  Goto state5
State3:
```