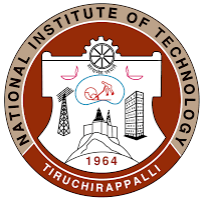


CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
~~5.~~	~~Names, Binding, Type Checking and Scopes~~
6.	Data Types
7.	Expressions and Assignment Statements
8.	**Statement-Level Control Structures**
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 8 – Statement-Level Control Structures

Objectives

- Overview of the evolution of control statements
- Thorough examination of selection statements
- Variety of looping statements
- Problems associated with unconditional branch statements
- Guarded command control statements

Introduction

- Computations in imperative-language programs are accomplished by evaluating expressions and assigning the resulting values to variables
- Two additional linguistic mechanisms are necessary to make the computations in programs flexible and powerful
    - Some means of selecting among alternative control flow paths (of statement execution)
    - Some means of causing the repeated execution of statements or sequences of statements
- Statements that provide these kinds of capabilities are called control statements
- Great deal of research and discussion was devoted to control statements between the mid-1960s and the mid-1970s
- One of the primary conclusions of these efforts was that, although a single control statement (a selectable goto) is minimally sufficient, a language that is designed *not* to include a goto needs only a small number of different control statements

Introduction

- It was proven that all algorithms that can be expressed by flowcharts can be coded in a programming language with only two control statements
  - One for choosing between two control flow paths *(switch, if)*
  - One for logically controlled iterations *(for, while, do)*
- Programmers care less about the results of theoretical research on control statements than they do about writability and readability
- Rather than requiring the use of a logically controlled loop statement for all loops, it is easier to write programs when a counter-controlled loop statement can be used to build loops that are naturally controlled by a counter
- Primary factor that restricts the number of control statements in a language is readability
- Too few control statements can require the use of lower-level statements, such as the goto, which also makes programs less readable

Introduction

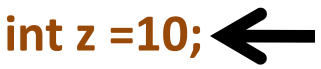
- How much a language should be expanded to increase its writability at the expense of its simplicity, size and readability
- A control structure is a control statement and the collection of statements whose execution it controls
- Only one design issue that is relevant to all of the selection and iteration control statements
    - Should the control structure have multiple entries?
- All selection and iteration constructs control the execution of code segments, and the question is whether the execution of those code segments always begins with the first statement in the segment
- It is now generally believed that multiple entries add little to the flexibility of a control statement, relative to the decrease in readability caused by the increased complexity
- Note that multiple entries are possible only in languages that include gotos and statement labels

Introduction

- Reader might wonder why multiple exits from control structures are not considered a design issue
- Reason is that all programming languages allow some form of multiple exits from control structures, the rationale being as follows
    - If all exits from a control structure are restricted to transferring control to the first statement following the structure, where control would flow if the control structure had no explicit exit, there is no harm to readability and also no danger
    - If an exit can have an unrestricted target and therefore can result in a transfer of control to anywhere in the program unit that contains the control structure, the harm to readability is the same as for a goto statement anywhere else in a program
- Languages that have a goto statement allow it to appear anywhere, including in a control structure
- Therefore, the issue is the inclusion of a goto, not whether multiple exits from control expressions are allowed

Miscellaneous

```
for(i = 0; i < n; i++)
{
  if (i > 10)
    break;
  else if (i > 5 && i < 10 && j = 0)
    break;
  else if (i > 0 and i < 5 && j =0)
    break;
  else
    i++;
}
int z =10;
```



```
# 60  for(i = 0; i < n; i++)
# 61  {
# 62    if (i > 10)
# 63      goto x;   // x can be in line # 10, 110, 120, ...
# 64    else if (i > 5 && i < 10 && j = 0)
# 65      goto y;   // y can be in line # 8, 1000, 1100, ...
# 66    else if (i > 0 and i < 5 && j =0)
# 67      goto z;   // z can be in line # 18, 810, 820, ...
# 68    else
# 69      i++;
# 70  }
# 71  int z =10;
# 72    .
      .
      .
```

Selection Statements

- Selection statement provides the means of choosing between two or more execution paths in a program
- Selection statements fall into two general categories
  - Two-way
  - n-way or multiple selection

Two-way Selection Statements

```
if control_expression
    then clause
    else clause
```

Design Issues

- What is the form and type of the expression that controls the selection?
- How are the then and else clauses specified?
- How should the meaning of nested selectors be specified?

Selection Statements

Control Expression

- Control expressions are specified in parentheses if the then reserved word (or some other syntactic marker) is not used to introduce the then clause

```
bool c = true;
if (c) { ... }          (or)    if c then
if (x > 5) { ... }      (or)    if x > 5 then
```

Clause Form

- In many contemporary languages, the then and else clauses appear as either single statements or compound statements

```
if (x > 5)
{
  ....;
  ....;
}
```

```
if (x > 5):
  ....;
  ....;
printf()
```

```
if (x > 5)
  ....;
  ....;
end if;
```

```
if (x > 5)
  ....;
  ....;
fi;
```

Selection Statements

Nesting Selectors

<if_stmt> → **if** <logic_expr> **then** <stmt>
| **if** <logic_expr> **then** <stmt> **else** <stmt>

- The issue was that when a selection statement is nested in the then clause of a selection statement, it is not clear to which if, an else clause should be associated

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

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

- Indentation seems to indicate that the else clause belongs with the first then clause
- In Java, as in many other imperative languages, the static semantics of the language specify that the else clause is always paired with the nearest previous unpaired then clause
- Static semantics rule, rather than a syntactic entity, is used to provide the disambiguation

Selection Statements

- C, C++, and C# have the same problem as Java with selectio statement nesting
- Perl requires that all then and else clauses be compound -> Does not have this problem

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

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

```
if a > b then
  sum = sum + a
  acount = acount + 1
else
  sum = sum + b
  bcount = bcount + 1
end
```

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

- Use of a special word for this purpose resolves the question of the semantics of nested selectors and also adds to the readability of the statement
  - Design of the selection statement in Fortran 95+ Ada, Ruby, and Lua
  - Because the *end* reserved word closes the nested if, it is clear that the else clause is matched to the inner then clause

Selection Statements

Ruby

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

Python

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

Multiple Selection Constructs

- Multiple-selection statement allows the selection of one of any number of statements or statement groups -> Generalization of a selector
- Need to choose from among more than two control paths in a program is common
- Multiple selector can be built from two-way selectors and gotos
    - Resulting structures are cumbersome, unreliable, and difficult to write and read
- Need for a special structure is clear

```
switch (expression)
{
    case constant1:
      // statements
      break;

    case constant2:
      // statements
      break;
    .
    .
    .
    default:
      // default statements
}
```

Multiple Selection Constructs

Design Issues

- Type of expression on which the selector is based
- Whether single statements, compound statements, or statement sequences may be selected
- Whether only a single selectable segment (*case*) can be executed when the statement is executed
- Form of the case value specifications
- What should result from the selector expression evaluating to a value that does not select one of the segments
    - Such a value would be unrepresented among the selectable segments
    - Simply disallow the situation from arising and have the statement do nothing at all when it does arise

Multiple Selection Constructs

Summary of Design Issues

- What is the form and type of the expression that controls the selection?
- How are the selectable segments specified?
- Is execution flow through the structure restricted to include just a single selectable segment?
- How are the case values specified?
- How should unrepresented selector expression values be handled, if at all?

Multiple Selection Constructs

Examples of Multiple Selectors

```
switch (expression) {
 case constant_expression1 : statement1 ;
 . . .
 case constantn : statement_n ;
 [default : statementn+1 ]
}
```

where the control expression and the constant expressions are some discrete type -> Integer Types, Character Types and Enumeration Types

- Selectable statements can be statement sequences, compound statements, or blocks
- Optional default segment is for unrepresented values of the control expression
- If the value of the control expression is not represented and no default segment is present, then the statement does nothing

Miscellaneous

```
enum coin{penny, nickel, dime, quarter, half-dollar, dollar};
enum coin money;
switch(money)
{
  case penny: printf("Penny");
              break;
  case nickel: printf("nickel");
              break;
}
```

0 1 2 3 4 5

Multiple Selection Constructs

- Switch statement does not provide implicit branches at the end of its code segment
- This allows control to flow through more than one selectable code segment on a single execution

```
switch (index) {
        case 1:
        case 3: odd += 1;
                sumodd += index;
        case 2:
        case 4: even += 1;
                sumeven += index;
        default: printf("Error in switch, index = %d\n", index);
}
```

This code prints the error message on every execution. Likewise, the code for the 2 and 4 constants is executed every time the code at the 1 or 3 constants is executed

- **break** statement, which is actually a restricted goto, is normally used for exiting switch statements

Multiple Selection Constructs

```
switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
          break;
  case 2:
  case 4: even += 1;
          sumeven += index;
          break;
  default: printf("Error in switch, index = %d\n", index);
}
```

The segments for the case values 1 and 2 are empty, allowing control to flow to the segments for 3 and 4, respectively

- Occasionally, it is convenient to allow control to flow from one selectable code segment to another
- Reliability problem with this design arises when the mistaken absence of a break statement in a segment allows control to flow to the next segment incorrectly
- Designers of C's switch traded a decrease in reliability for an increase in flexibility

Multiple Selection Constructs

- C switch statement has virtually no restrictions on the placement of the case expressions, which are treated as if they were normal statement labels
- This laxness can result in highly complex structure within the switch body

```
switch (x)
        default:
        if (prime(x))
                case 2: case 3: case 5: case 7:
                        process_prime(x);
        else
                case 4: case 6: case 8: case 9: case 10:
                        process_composite(x);
```

Multiple Selection Constructs

C#

```
switch (value) {
   case -1:
      Negatives++;
      break;
   case 0:
      Zeros++;
      goto case 1;
   case 1:
      Positives++;
   default:
      Console.WriteLine("Error in switch \n");
}
```

- Rule is that every selectable segment must end with an explicit unconditional branch statement
  - break -> Transfers control out of the switch statement
  - goto -> Can transfer control to one of the selectable segments (or virtually anywhere else)
- Control expression and the case statements can be strings

Multiple Selection using *if*

- In many situations, a switch or case construct is inadequate for multiple selection
- Eg: When selections must be made on the basis of a boolean expression rather than some ordinal type, nested two-way selectors can be used to simulate a multiple selector
- Notice that this example is not easily simulated with a switch statement, because each selectable statement is chosen on the basis of a Boolean expression
- Else-if construct is not a redundant form of switch

```
if count < 10:
      Bag1 = True;
elif count < 100:
      Bag2 = True;
elif count < 1000:
      Bag3 = True;
```

Iterative Statements

- Iterative statement is one that causes a statement or collection of statements to be executed zero, one, or more times
- An iterative statement is often called a loop
- If some means of repetitive execution of a statement or collection of statements were not possible, programmers would be required to state every action in sequence
- Useful programs would be huge and inflexible and take unacceptably large amounts of time to write and mammoth amounts of memory to store
- First iterative statements in programming languages were directly related to arrays
- Primary categories are defined by how designers answered two basic design questions
    - How is the iteration controlled?
    - Where should the control mechanism appear in the loop statement?

Iterative Statements

- Primary possibilities for iteration control are logical, counting, or a combination of the two
- Main choices for the location of the control mechanism are the top of the loop or the bottom of the loop
    - Issue is not the physical placement of the control mechanism
    - Rather, it is whether the mechanism is executed and affects control before or after execution of the statement's body
- Body of an iterative statement is the collection of statements whose execution is controlled by the iteration statement
- Pretest -> Test for loop completion occurs before the loop body is executed
- Posttest -> Occurs after the loop body is executed
- Iteration statement and the associated loop body together form an iteration statement

Iterative Statements

Counter Controlled Loops

- Counting iterative control statement has a variable, called the loop variable, in which the count value is maintained
- It also includes some means of specifying the initial and terminal values of the loop variable and the difference between sequential loop variable values, often called the stepsize. Eg: *for(int i = 0; i < 9; i++)*
    - Initial, Terminal and Stepsize specifications of a loop are called the loop parameters

Design Issues

- Type of the loop variable and that of the loop parameters obviously should be the same or at least compatible, but what types should be allowed?
    - One apparent choice is integer, but what about enumeration, character and floating-point types?

Iterative Statements

- Whether the loop variable is a normal variable, in terms of scope, or whether it should have some special scope
- Allowing the user to change the loop variable or the loop parameters within the loop can lead to code that is very difficult to understand, so another question is whether the additional flexibility that might be gained by allowing such changes is worth that additional complexity
- A similar question arises about the number of times and the specific time when the loop parameters are evaluated
    - If they are evaluated just once, it results in simple but less flexible loops

Iterative Statements

Summary of these design issues

- What are the type and scope of the loop variable?
- Should it be legal for the loop variable or loop parameters to be changed in the loop, and if so, does the change affect loop control?
- Should the loop parameters be evaluated only once, or once for every iteration?

The for Statement of the C-Based Languages

```
for (expression_1 ; expression_2 ; expression_3)
  loop body
```

- Loop body can be a single statement, a compound statement, or a null statement
- First expression is for initialization and is evaluated only once, when the for statement execution begins
- Second expression is the loop control and is evaluated before each execution of the loop body
  - Zero value means false and all nonzero values mean true
  - If the value of the second expression is zero, then for is terminated
  - Otherwise, the loop body statements are executed
- In C99, the expression also could be a Boolean type
- Last expression in the for is executed after each execution of the loop body
  - It is often used to increment the loop counter

The for Statement of the C-Based Languages

```
    expression_1
loop:
    if expression_2 = 0 goto out
    [loop body]
    expression_3
    goto loop
out: . . .
```

```
for (count = 1; count <= 10; count++)
  . . .
}
```

- All of the expressions of C's for are optional
- An absent second expression is considered true, so a for without one is potentially an infinite loop
- If the first and/or third expressions are absent, no assumptions are made
- **Eg:** If the first expression is absent, it simply means that no initialization takes place

The for Statement of the C-Based Languages

- C for design choices are the following
  - There are no explicit loop variables or loop parameters
  - All involved variables can be changed in the loop body
  - Expressions are evaluated in the order stated previously

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

```
    count1 = 0
    count2 = 1.0
loop:
    if count1 > 10 goto out
    if count2 > 100.0 goto out
    count1 = count1 + 1
    sum = count1 + count2
    count2 = count2 * 2.5
    goto loop
out: ...
```

- Whole expression is evaluated, but the resulting value is not used in the loop control

The for Statement of the C-Based Languages

- for statement of C99 and C++ differs from that of earlier versions of C in two ways
  - First, in addition to an arithmetic expression, it can use a Boolean expression for loop control
  - Second, the first expression can include variable definitions

```
for (int count = 0; count < len; count++) { . . . }
```

- Scope of a variable defined in the for statement is from its definition to the end of the loop body
- In all of the C-based languages, the last two loop parameters are evaluated with every iteration
- Furthermore, variables that appear in the loop parameter expression can be changed in the loop body
- Therefore, these loops can be far more complex and are often less reliable than the counting loop of Ada

Logically Controlled Loops

- Collections of statements must be repeatedly executed, but the repetition control is based on a Boolean expression rather than a counter
- Logically controlled loop is convenient
- Logically controlled loops are more general than counter-controlled loops
- Every counting loop can be built with a logical loop, but the reverse is not true
- Recall that only selection and logical loops are essential to express the control structure of any flowchart

Design Issues

- Should the control be pretest or posttest?
- Should the logically controlled loop be a special form of a counting loop or a separate statement?

Logically Controlled Loops

Examples

- C-based programming languages include both pretest and posttest logically controlled loops that are not special forms of their counter-controlled iterative statements

```
while (control_expression)
   loop body
```

```
do
   loop body
while (control_expression);
```

- In the pretest version of a logical loop (while), the statement or statement segment is executed as long as the expression evaluates to true
- In the posttest version (do), the loop body is executed until the expression evaluates to false
- Only real difference between the do and the while is that the do always causes the loop body to be executed at least once. In both cases, the statement can be compound

Logically Controlled Loops

while

```
loop:
  if control_expression is false goto out
  [loop body]
  goto loop
out: . . .
```

do-while

```
loop:
  [loop body]
  if control_expression is true goto loop
```

- It is legal in both C and C++ to branch into both while and do loop bodies
- C89 version uses an arithmetic expression for control
- C99 and C++ -> It may be either arithmetic or Boolean

User-Located Loop Control Mechanisms

- Sometimes, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop body
- Some languages provide this capability
- Such loops have the structure of infinite loops but include user-located loop exits
- Question is whether a single loop or several nested loops can be exited

Design Issues

- Should the conditional mechanism be an integral part of the exit?
- Should only one loop body be exited, or can enclosing loops also be exited?

User-Located Loop Control Mechanisms

- C, C++, Python, Ruby, and C# have unconditional unlabeled exits (break)

```
outerLoop:
  for (row = 0; row < numRows; row++)
    for (col = 0; col < numCols; col++) {
      sum += mat[row][col];
      if (sum > 1000.0)
        break outerLoop;
    }
```

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) continue;
  sum += value;
}
```

- C, C++ and Python include an unlabeled control statement, continue, that transfers control to the control mechanism of the smallest enclosing loop
- This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop structure
- A negative value causes the assignment statement to be skipped, and control is transferred instead to the conditional at the top of the loop

User-Located Loop Control Mechanisms

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) break;
  sum += value;
}
```

- A negative value terminates the loop
- Break provide for multiple exits from loops, which may seem to be somewhat of a hindrance to readability
- However, unusual conditions that require loop termination are so common that such a statement is justified
- Furthermore, readability is not seriously harmed, because the target of all such loop exits is the first statement after the loop (or an enclosing loop) rather than just anywhere in the program
- The motivation for user-located loop exits is simple -> They fulfill a common need for goto statements through a highly restricted branch statement
- The target of a goto can be many places in the program, both above and below the goto itself
- However, the targets of user-located loop exits must be below the exit and can only follow immediately the end of a compound statement

Iteration Based on Data Structures

- A Do statement in Fortran uses a simple iterator over integer values

  Do Count = 1, 9, 2

- Python -> for count in range [0, 9, 2]:
- A general data-based iteration statement uses a user-defined data structure and a user-defined function (the iterator) to go through the structure's elements
- Iterator is called at the beginning of each iteration, and each time it is called, the iterator returns an element from a particular data structure in some specific order
- Eg: Suppose a program has a user-defined binary tree of data nodes, and the data in each node must be processed in some particular order
- A user-defined iteration statement for the tree would successively set the loop variable to point to the nodes in the tree, one for each iteration

Iteration Based on Data Structures

- Initial execution of the user-defined iteration statement needs to issue a special call to the iterator to get the first tree element
- Iterator must always remember which node it presented last so that it visits all nodes without visiting any node more than once -> Iterator must be history sensitive
- A user-defined iteration statement terminates when the iterator fails to find more elements
- If the tree root is pointed to by a variable named root, and if traverse is a function that sets its parameter to point to the next element of a tree in the desired order

```
for (ptr = root; ptr == null; ptr = traverse(ptr)) {
  ...
}
```

Unconditional Branching

- Unconditional branch statement transfers execution control to a specified location in the program
- Unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program's statements
- However, using the goto carelessly can lead to serious problems
- Goto has stunning power and great flexibility (all other control structures can be built with goto and a selector), but it is this power that makes its use dangerous
- Without restrictions on use, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly unreliable and costly to maintain
- These problems follow directly from a goto's capability of forcing any program statement to follow any other in execution sequence, regardless of whether that statement precedes or follows the previously executed statement in textual order

Unconditional Branching

- Readability is best when the execution order of statements is nearly the same as the order in which they appear—in our case, this would mean top to bottom, which is the order with which we are accustomed
- Restricting gotos so they can transfer control only downward in a program partially alleviates the problem
- It allows gotos to transfer control around code sections in response to errors or unusual conditions but disallows their use to build any sort of loop
- A few languages have been designed without a goto. Eg: Java, Python, and Ruby
- Languages that have eliminated the goto have provided additional control statements, usually in the form of loop exits, to code one of the justifiable applications of the goto

Guarded Commands

- Methodology is described in Dijkstra (1976)
- Nondeterminism is sometimes needed in concurrent programs
- A selectable segment of a selection statement in a guarded-command statement can be considered independently of any other part of the statement, which is not true for the selection statements of the common programming languages
- Dijikstra's selection statement has the form

```
if <Boolean expression> -> <statement>
[] <Boolean expression> -> <statement>
[] ...
[] <Boolean expression> -> <statement>
fi
```

- Small blocks, called *fatbars, are used to separate the guarded clauses and allow the* clauses to be statement sequences
- Each line in the selection statement, consisting of a Boolean expression (a guard) and a statement or statement sequence, is called a guarded command

Guarded Commands

- This selection statement has the appearance of a multiple selection, but its semantics is different
- All of the Boolean expressions are evaluated each time the statement is reached during execution
- If more than one expression is true, one of the corresponding statements can be nondeterministically chosen for execution

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

- If i = 0 && j > i, then it chooses nondeterministically between the first and third assignment statements
- If i is equal to j and is not zero, a runtime error occurs because none of the conditions is true

- An implementation may always choose the statement associated with the first Boolean expression that evaluates to true
- But it may choose any statement associated with a true Boolean expression
- So, the correctness of the program cannot depend on which statement is chosen (among those associated with true Boolean expressions)
- If none of the Boolean expressions is true, a run-time error occurs that causes program termination
- This forces the programmer to consider and list all possibilities

Guarded Commands

- To find the largest of two numbers

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

- This computes the desired result without overspecifying the solution
- In particular, if x and y are equal, it does not matter which we assign to max
- This is a form of abstraction provided by the nondeterministic semantics of the statement

```
if (x >= y)
  max = x;
else
  max = y;
```

```
if (x > y)
  max = x;
else
  max = y;
```

- This choice between the two statements complicates the formal analysis of the code and the correctness proof of it
- This is one of the reasons why guarded commands were developed by Dijkstra

Guarded Commands

```
do <Boolean expression> -> <statement>
 [] <Boolean expression> -> <statement>
 [] . . .
 [] <Boolean expression> -> <statement>
od
```

- If more than one is true, one of the associated statements is nondeterministically (perhaps randomly) chosen for execution, after which the expressions are again evaluated
- When all expressions are simultaneously false, the loop terminates
- Given four integer variables, q1, q2, q3, and q4, rearrange the values of the four so that q1 ≤ q2 ≤ q3 ≤ q4
- Without guarded commands, one straightforward solution is to put the four values into an array, sort the array, and then assign the values from the array back into the scalar variables q1, q2, q3, and q4

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
 [] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
 [] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

Miscellaneous

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

- Let, q1 = 4; q2 = 3; q3 = 1; q4 = 2

Iteration 1

q1 > q2 -> True; q2 > q3 -> True; q3 > q4 -> False

- q1 > q2 -> temp = 4; q1 = 3; q2 = 4 // q1 = 3; q2 = 4; q3 = 1; q4 = 2

Iteration 2

q1 > q2 -> False; q2 > q3 -> True; q3 > q4 -> False

- q2 > q3 -> temp = 4; q2 = 1; q3 = 4 // q1 = 3; q2 = 1; q3 = 4; q4 = 2

Iteration 3

q1 > q2 -> True; q2 > q3 -> False; q3 > q4 -> True

- q1 > q2 -> temp = 3; q1 = 1; q2 = 3 // q1 = 1; q2 = 3; q3 = 4; q4 = 2

Iteration 4

q1 > q2 -> False; q2 > q3 -> False; q3 > q4 -> True

- q3 > q4 // q1 = 1; q2 = 3; q3 = 2; q4 = 4

Iteration 5

q1 > q2 -> False; q2 > q3 -> True; q3 > q4 -> False

- q2 > q3 // q1 =1; q2 = 2; q3 = 3; q4 = 4

Iteration 6

- q1 > q2 -> False; q2 > q3 -> False; Q3 > q4 -> False // q1 = 1; q2 = 2; q3 = 3; q4 = 4

Guarded Commands

- Dijkstra's guarded command control statements are interesting, in part because they illustrate how the syntax and semantics of statements can have an impact on program verification and vice versa
- Program verification is virtually impossible when goto statements are used
- Verification is greatly simplified if
  - Only logical loops and selections are used
  - Only guarded commands are used
- It should be obvious, however, that there is considerably increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts

Thank You