Statement level control structures

CS 315 – Programming Languages

Pinar Duygulu
Bilkent University

Control Statements: Evolution

- 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

Control structure

- **Control Structure**:
- a control statement
- And the collection of statements whose execution it controls.
- **Control Statements** are:
- **Conditional** execution statements
- **Repeated** execution statements

- Control structures should have **single entry** and **single exit** points.

Compound statements

- ALGOL60 introduced the first statement collection structure.
- First introduced in ALGOL 60 in the form of
- begin
- statement$_1$
- ...
- statement$_n$
- end
- A collection of statements is treated **as a single statement**.
- ALGOL60 allows **data declarations** at the beginning of a compound statement making it a **block**.
- begin
- integer v1, v2;
- ...
- end
- **Scope** of `v1` and `v2` **is** the **block**.

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

Two-Way Selection Statements

- General form:

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

- 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?

Selection Statements

- A special case: **Single-way selector**: In most cases a subform of a two-way selector. FORTRAN IV has no two-way selector.
- In **FORTRAN** this is called **logical IF**
- FORTRAN: **`IF`** (boolean_expr) statement
- Problem: can select only a single statement; to select more, a **`GOTO`** must be used, as in the following example

```
  IF (.NOT. condition) GOTO 20
...
  20 CONTINUE
```

- Negative logic is bad for readability
- This problem was solved in FORTRAN 77

Selection Statements

- Example,
- IF (A .GT. B) GOTO 10
- A = B
- C = A + D
- 10 CONTINUE
- ...
- IF (A .GT. 0) A = 0

Selection Statements

- Most later languages allow compounds for the selectable segment of their single-way selectors

- The concept of **compound statements** is introduced by **ALGOL 60**.
- **ALGOL60** allows compound statements in selection.
- **if** (*Boolean Expression*) **then**
- begin
- statement$_1$
- ...
- statement$_n$
- end

Two-Way Selection: Examples

- ALGOL60 introduced the first **two-way selector**.
- **`if`** (boolean_expr)
  `then` statement (then clause)
  `else` statement (else clause)
- The statements could be single or compound

Nesting selectors

- Java example

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

- Which `if` gets the `else`?   AMBIGUOUS

Nesting selectors

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

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

Nesting selectors

- **PASCAL, C, C++, Java**: match else to the nearest unmatched if.
  - Only a semantic rule, not a syntactic rule (not clear from the syntax).
- **Perl** requires all **then** and **else** statements to be **compound**.
- In **ALGOL60**, an if statement is **not allowed** to be **nested directly in** a **then** clause. So, *the code above is illegal*.
  - The programmer must use compound statements.

Special Words and Selection Closure

- Another solution to the problem above is to use **special words**.
- **IF** marks the beginning.
- **THEN** marks the end of condition and the beginning of then part.
- **ELSE** marks the end of then part and the beginning of else part.
- **END** or **END IF** marks the end of **IF**.
- In **Modula-2**, If-then-else construct has an **END**, even if a single stmt.

If *cond*	If *cond*
THEN	THEN
ts1	ts1
...	...
tsn	tsn
ELSE	END
es1	
...	
Esm	
END	

- In **FORTRAN77**, **FORTRAN90** and **Ada**, the **closing** of an **`IF`** is **`END IF`**.
- This is even more **readable** than Modula-2.

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
- 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. What is done about unrepresented expression values?

Multiple-Way Selection Statements

- Early multiple selectors:
  - FORTRAN arithmetic `IF` (a three-way selector)

```
IF (arithmetic expression) N1, N2, N3
```

- If the value of the arithmetic expression is less than 0 go to the statement with label *N1*, if it is 0 go to the statement with label *N2*, if it is more than 0 go to *N3*.

```
print *, "Enter a number"
   read *, X
   if (X-1) 10, 20, 30
10 print *, X, " is less than 1"
   go to 40
20 print *, X, " is equal to 1"
   go to 40
30 print *, X, " is more than 1"
40 end
```

NOT READABLE

  - Segments require `GOTO`s
  - Not encapsulated (selectable segments could be anywhere)

Multiple-Way Selection Statements

- FORTRAN's computed GOTO
- GOTO (*label*$_1$, *label*$_2$, ..., *label*$_n$) *integer_expression*
- Evaluate *integer_expression*. If it is ***i***, go to the statement with label *label*$_i$.
- If no label for the value of the expression is given go to the next statement.

Multiple-Way Selection Statements

- Modern Multiple Selectors
- **case** statement in **ALGOL-W**,
- **case** *integer_expression* **of**
- begin
- $statement_1$;
- ...
- $statement_n$;
- end
- If `integer_expression` is ***i***, evaluate $\texttt{statement}_i$.

Multiple-Way Selection Statements

- **case** statement in **PASCAL** (selectable statements are labeled)
- **case** *ordinal_type_expression* **of**
- *constant_list_1*: *statement_1*;
- ...
- *constant_list_n*: *statement_n*;
- end
- A constant may not appear in more than one constant list.
- **Undefined** results occur if the value of the expression is not in any of the lists (standard Pascal).
- ANSI/IEEE Pascal Standard specifies that the code generated should detect such cases and **report error messages**.
- Many dialects of PASCAL now include **else** to match any **unlisted** value.

Multiple-Way Selection Statements

- Example:

```
Case idx of
1,3: a := a+1;
2: begin
        b := b+1;
        c := c-1;
      end;
  else
    writeln ("idx: ", idx, " is strange.");
  end;
```

Multiple-Way Selection Statements

switch in **C**: relatively primitive

switch (*expression*){/* expression is integer type */
case *constant_expression_1*: *statement_1*;
...
case *constant_expression_n*: *statement_n*;
[**default**: **statement_n+1**;]
}

Control of execution is transferred to the statement whose *constant expression* is equal to the *expression*. Then all following cases are executed.

break statement is used to avoid the execution of unwanted cases.

Multiple-Way Selection Statements

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)**

Multiple-Way Selection Statements

- **The Ada `case` statement**

```
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` statements.

Multiple-Way Selection Statements

- **Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses,**

```
if ...
    then ...
elsif ...
   then ...
elsif ...
   then ...
   else ...
end if
```

Multiple-Way Selection Statements

- **Ada** provides a special case of **nested if** along with a **case statement**.
- **if** *boolean_expression_1* **then** *statement_1*;
- **elsif** boolean_expression_2 **then** *statement_2*;
- **elsif** boolean_expression_3 **then** *statement_3*;
- ...
- **end if**;
- This is **more readable** than **standard nested if** and **case** structures.

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?
   logical
   counting
   a combination of the two
2. Where is the control mechanism in the loop?
   pretest: at the top or
   posttest: at the bottom

Counter controlled loops

- Control statement has a variable called **loop variable**, along with its **initial** and **terminal** values, and sometimes a **stepsize**, they are called **loop parameters**.
    - Design Issues
    - **type** and **scope** of the loop variable?
    - **value** of the loop variable at the loop **termination**?
    - can the **loop variable** or loop parameters be **changed** in the loop?
    - **pretest** or **posttest**?
    - loop parameters are evaluated only once or for every iteration?

Iterative statements : Examples

The DO Statement of FORTRAN77 and FORTRAN 90

DO *label variable* = *initial*, *terminal* [, *stepsize*]

Loop variable can be **integer**, **real** or **double-precision**.

DO 10 R=0.5,9.9,0.1

Iteration count is computed **before** the execution.

Parameters can be changed in the loop, but iteration count remains unchanged.

Gnu implantation of FORTRAN 77 does not allow the loop counter to be modified in the loop. Following code does not compile in g77.

```
    Do 10 J=1,10
    print *, J
    J = 3
10 CONTINUE
```

It is compiled by f77 (SUN compiler), prints infinite 5's (infinite loop).

IN FORTRAN space character is not used as a token separator. That is, `DO 10 J=1,10`, and `DO10J=1,10` are the same.

If you type "`.`" for "`,`" it is `DO10J=1.10`, an assignment.

Iterative statements : Examples

FORTRAN 90 DO includes

[name] DO *variable* = *initial*, *terminal* [, *stepsize*]

...

END DO [name]

Fortran DO statement is **pretest**.

The following program produces no output.

```
   DO 20 I =4,3
   Print *, I
20 continue
end
```

However, the following program

```
   DO 20 I =4,3,-1
20 Print *, I
end
```

Produces the following output:

4

3

Iterative statements : Examples

The ALGOL60 for Statement

A significant generalization of FORTRAN's DO: User can **combine** a **counter** and a **boolean expression** for the loop control.

However, **Flexibility** leads to **complexity**.

Syntax in EBNF:

```
<for_stmt>-> for var := <list_elt>{,<list_elt>} do <statement>
<list_elt> -> <expression>
           | <expression> step <expression> until <expression>
           | <expression> while <Boolean_expression>
```

Combines a **counter** and a **Boolean expression** for loop control.

Example, the following statements are equivalent.

for i := 1,2,3,4,5 **do** list[i] := 0

for i := 1 **step** 1 **until** 5 **do** list[i] := 0

for i := 1, i+1 **while** (i <= 5) **do** list[i] := 0

Much **more complex**, yet **more flexible**, then any other for loop.

Iterative statements : Examples

```
Code for step-until
for_var <- init_expr
loop:
form
  Until <- until_exp
  step <- step_exp
  tmp <- (for_var - until) * SIGN(step)
  if tmp > 0 go to out
  [loop body]
  for_var <- for_var + step
  go to loop
out: ...
```

Iterative statements : Examples

- Example,
- **for** i := 1, 2, 5 **step** 2 **until** 11, 2*i **while** i<90, 41, -5 **do**
- will execute for the following values of `i`:
- 1, 2, `5, 7, 9, 11`, `26, 52,` 41, -5
- Too difficult to understand
- All expressions are evaluated for every iteration
- pretest loop
- Loop variables can be integer or real
- Loop variables can not be changed in the loop body

Iterative statements : Examples

```
<for_stmt>-> for var := <list_elt>{,<list_elt>} do <statement>
<list_elt> -> <expression>
           | <expression> step <expression> until <expression>
           | <expression> while <Boolean_expression>
```

for i := 1, 2, 5 **step** 2 **until** 11, 2*i **while** i<90, 41, -5 **do**

will execute for the following values of `i`:

1, 2, 5, 7, 9, 11, 26, 52, 41, -5

Iterative statements : Examples

Example,
i := 1;
for c := 1 step c until 3*i do i:=i+1

will execute as

c ←1

until←3*1=3	until←3*i=6	until←9	until←12	until←15
i ←i+1=2	i ←2+1=3	i ←3+1=4	i ←4+1=5	(16-15)>0
c ←c+c=2	c ←c+c=4	c ←c+c=8	c ←16	**Terminate**

Iterative statements : Examples

The Pascal for Statement

The model of simplicity

for <var> := <init_exp> (**to** | **down to**) <final_exp> **do** <stmt>

Iterative statements : Examples

The Ada for Statement
Relatively **simple**, **pretest** loop.

for *variable* **in** [**reverse**] *discrete_range* **loop**
...
end loop

Scope of the variable is the range of the loop.

```
SUM : FLOAT := 0;
COUNT : FLOAT := 1.35;
for COUNT in 1..10 loop
   SUM := SUM + COUNT      this COUNT is INTEGER
end loop                   FLOAT COUNT is hidden here
```

here `COUNT` *is* `1.35, SUM is 55`

Iterative statements : Examples

- C's `for` statement

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

- The expressions can be whole statements, or even statement sequences, with the statements separated by commas
  - The value of a multiple-statement expression is the value of the last statement in the expression
- There is no explicit loop variable
- Everything can be changed in the loop
- The first expression is evaluated once, but the other two are evaluated with each iteration

Iterative statements : Examples

The for Statement of C, C++, and Java

- pretest

for (exp1; exp2; exp3) statement

Statement can be **single**, **compound** or **null**.
`exp1` is evaluated **only once** when execution **begins**.
`exp2` is evaluated **before each** execution of the loop.
loop terminates when exp2 is 0
`exp3` is evaluated **after each** execution of the loop.

```
exp1
loop:
 if exp2 = 0 go to out
 [loop body]
 exp3
 go to loop
out: ...
```

Iterative statements : Examples

- All of the **expressions** (`exp`’s) of C’s `for` are **optional**.
- So, `for(;; ) stmt`; is legal.
- An **absent `exp2`** is considered **true**.
- No explicit loop variable or parameters
- Variables can be changed in the body
- Each expression can comprise **multiple statements** separated by `,` the value of the expression is the value of the last statement.

Iterative Statements: Examples

- 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)
- Java and C#
  - Differs from C++ in that the control expression must be Boolean

Iterative Statements: Logically-Controlled Loops

- More general than counter-controlled loops
- Repetition control is based on a Boolean
- Design issues:
  - Pre-test or post-test?
  - Should the logically controlled loop be a special case of the counting loop statement ? expression rather than a counter
- General forms:

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

Iterative Statements: Logically-Controlled Loops

- Pascal has separate pre-test and post-test logical loop statements (`while-do` and `repeat-until`)
  - Pascal **repeat-until** (posttest) loop can have a single statement, compound statement, or statement sequence. This is the only control structure with this flexibility. This another reason for **lack of orthogonality** in Pascal.
- C and C++ also have both, but the control expression for the post-test version is treated just like in the pre-test case (`while-do` and `do- while`)
- Java is like C, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no **goto**

Iterative Statements: Logically-Controlled Loops

- Ada has a pretest version, but no post-test
- FORTRAN 77 and 90 have neither
- Perl has two pre-test logical loops, `while` and `until`, but no post-test logical loop

Iterative Statements: User-located loop control mechanisms

- 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?

Iterative Statements: User-Located Loop Control Mechanisms `break` and `continue`

- C , C++, and Java: **`break`** statement
- Unconditional; for any loop or **`switch`**; one level only
- Java and C# have a labeled `break` statement: control transfers to the label
- An alternative: **`continue`** statement; it skips the remainder of this iteration, but does not exit the loop

Iterative Statements: User-Located Loop Control Mechanisms

Both **FORTRAN90** and **Ada** have loops with **no control** (infinite loop).

Ada example,

loop
...
end loop

and an **exit** statement is **provided**.

exit [*loop-label*] [**when** *condition*]

Iterative Statements: User-Located Loop Control Mechanisms

- In **Ada**, loops can have labels.
- If *loop-label* is omitted, exit statement **causes** the **termination** of **only** the **block** in which **it appears**.
- Transfers control to the statement after the loop.

```
OL: loop
   ...
   IL: for ... loop
     ...
     exit OL when x>0;            (exits the OL loop)
     ...
   end loop IL;
...
end loop OL;
```

Iterative Statements: User-Located Loop Control Mechanisms

C and **FORTRAN90** have statements to **skip** the rest of a single iteration.

In **C**, **`continue`**, in **FORTRAN 90** **`CYCLE`**.

```
while ( ... ) {
   ...
  if (   )                continue;
  ...                         |
}   <-------------------------|
```

Such statements **make** a language **unreadable**.

Iterative Statements: Iteration Based on Data Structures

- Number of elements of in a data structure control loop 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
- C's **`for`** can be used to build a user-defined iterator:

```
for (p=root; p==NULL; traverse(p)){
  }
```

Iterative Statements: Iteration Based on Data Structures

- 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
- Well-known mechanism: `goto` statement
- Major concern: Readability
- Some languages do not support `goto` statement (e.g., Module-2 and Java)
- C# offers `goto` statement (can be used in `switch` statements)
- Loop exit statements are restricted and somewhat camouflaged `goto`'s

Guarded Commands

- Suggested by Dijkstra
- Purpose: to support a new programming methodology that supported verification (correctness) during development
- Basis for two linguistic mechanisms for concurrent programming (in CSP and Ada)
- Basic Idea: if the order of evaluation is not important, the program should not specify one

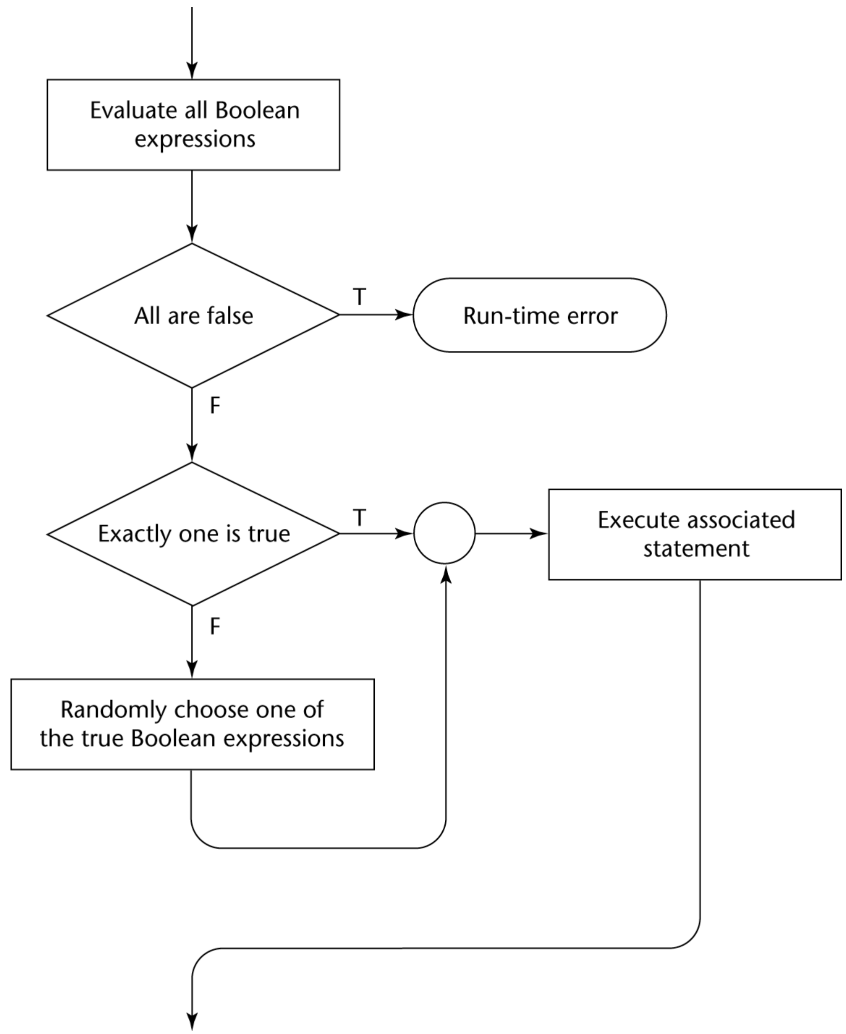
Selection Guarded Command

- Form

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

- Semantics: when construct is reached,
  – Evaluate all Boolean expressions
  – If more than one are true, choose one non-deterministically
  – If none are true, it is a runtime error

Selection Guarded Command: Illustrated



Selection Guarded Command

- To find the largest of the two values:
- if x >= y -> max := x
- [] y >= x -> max := y
- fi
- A good way to state that the **order** of execution **is irrelevant**.

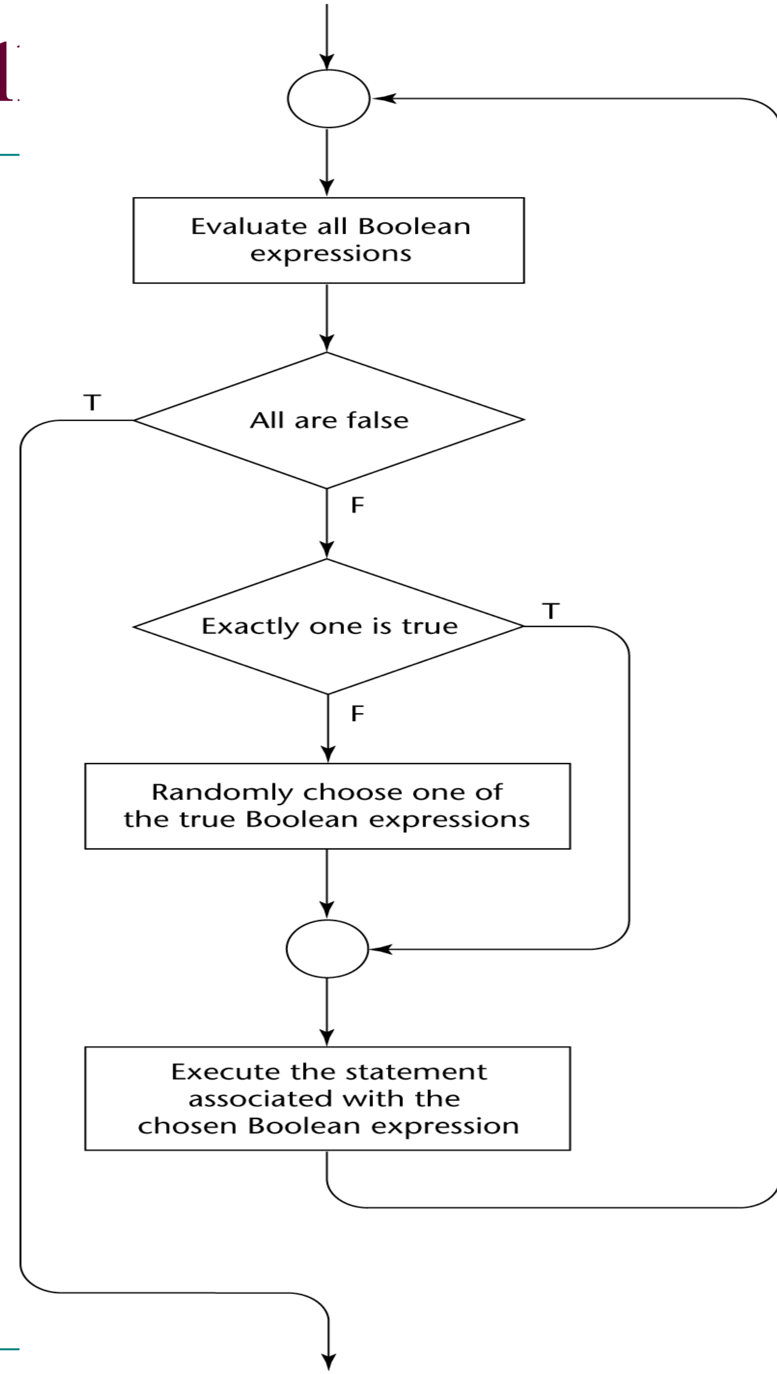
Loop Guarded Command

- **Form**

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

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - If none are true, exit loop

Loop Guarded Command: Il



Guarded Commands: Rationale

- Connection between control statements and program verification is intimate
- Verification is impossible with `goto` statements
- Verification is possible with only selection and logical pretest loops
- Verification is relatively simple with only guarded commands