STATEMENT LEVEL CONTROL STRUCTURES
Levels of Control Flow

- Within expressions (Chapter 7)
- Among program units
  - Function calls
- Among program statements
  - Selection
  - Repetition
  - Unconditional branching
Control Structure

- A *control structure* is a control statement and the statements whose execution it controls.
- Fortran control statements were based on the underlying hardware:
  - Branch instructions based on a test
  - Unconditional jump instructions
- Design question
  - Should a control structure have multiple entries?
Control Statements: Evolution

- Theory says
  - All algorithms represented by flowcharts can be coded with only two-way selection and pretest logical loops

- Having multiple kinds of control statements makes things easier for the programmer
Selection Statements

- A *selection statement* provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors (if)
  - Multiple-way selectors (switch)
Two-Way Selection Statements

- **General form:**
  
  ```
  if control_expression
    then-clause
  else
    else-clause
  ```

- **Control expression should evaluate to true or false**
- **Else-clause can usually be omitted**
- **Either the then-clause or the else-clause gets executed**
Two-Way Selection 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?
The 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.

• In C89, C99, Python, and C++, the control expression can be an arithmetic expression.

• In languages such as Ada, Java, Ruby, and C#, the control expression must be Boolean.
Clause Form

- In many contemporary languages, the *then* and *else* clauses can be single statements or compound statements
- In Fortran 95, Ada, and Ruby, clauses are statement sequences
- Python uses indentation to define clauses

```python
if x > y :
    x = y
    print "case 1"
```
Syntax for if

- In Java et al.
  \[<\text{ifstmt}> \rightarrow \text{if} (<\text{test}>) <\text{stmt}> \]
  \[\mid \text{if} (<\text{test}>) <\text{stmt}> \text{ else } <\text{stmt}>\]

- Derive
  \[\text{if} (p) \text{ if} (q) \text{ s1 else s2}\]
  - This grammar is ambiguous

- How to fix it?
  - semantic rule (else goes with *nearest* if)
  - keyword at end (Ruby)
Dangling else

- Java example
  
  ```java
  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

- 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#
Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
switch for Multiple Selection

```java
switch (expression) {
    case const_expr_1: stmt_1;
    ...
    case const_expr_n: stmt_n;
    [default: stmt_n+1]
}
```
Multiple-Way Selection: C, C++, and Java

\[
\text{switch (expression) } \\
\quad \text{case const\_expr\_1: stmt\_1; } \\
\quad \ldots \\
\quad \text{case const\_expr\_n: stmt\_n; } \\
\quad \text{[default: stmt\_n+1]}
\]
switch in C

- **Design choices**
  
  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: C#

• Differs from C in that it 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* or *break*)

• Also, in C# the control expression and the case constants can be *strings*
Multiple-Way Selection: 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)
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 (\(\mid\))
Multiple-Way Selection: Ruby

• Two forms for case statements
  1. One form uses *when* conditions

```ruby
leap = case
  when year % 400 == 0 then true
  when year % 100 == 0 then false
  else year % 4 == 0
end
```

2. The other uses a *case* value and *when* values

```ruby
case in_val
  when -1 then neg_count++
  when 0 then zero_count++
  when 1 then pos_count++
  else puts "Error – in_val is out of range"
end
```
Multiple-Way Selection Using `if`

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

- Example: Python
  ```python
  if count < 10 :
      bag1 = True
  elif count < 100 :
      bag2 = True
  elif count < 1000 :
      bag3 = True
  ```
if–else chain: perl, ruby

if ...
    {. . . }
elsif ...
    {. . . }
elsif ...
    {. . . }
elsif ...
    {. . . }
else { ... }
Multi-way Selection in Fortran

- **Arithmetic if (three-way selector)**
  
  ```fortran
  if (arithmetic-expr) N1, N2, N3
  - N1, N2, N3 are statement labels
  - Segments require GOTO statements
  - Selectable statements can be anywhere
  ```

- **Computed GOTO**
  
  ```fortran
  goto (N1, N2, N3, ...) arithmetic-expr
  ```
Multiple-Way Selection: Scheme

- **cond** special form
  
  ```lisp
  (cond
    (test1 result1) ...
    (testn resultn)
    (else defaultResult))
  ```

- **case** special form
  
  ```lisp
  (case expr
    ((valuelist1) result1) ...
    (valuelistn) resultn)
    (else defaultResult))
  ```