Software II: Principles of Programming Languages

Lecture 8 – Statement-Level Control Structures

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

- A ***control structure*** is a control statement and the statements whose execution it controls
- Design question
  - Should a control structure have multiple entries?

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?

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 arithmetic
- In most other languages, 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 Perl, all clauses must be delimited by braces (they must be compound)
- In Fortran 95, Ada, Python, and Ruby, clauses are statement sequences
- Python uses indentation to define clauses

```
if x > y :
      x = y
      print " x was greater than y"
```

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 previous **`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#

Nesting Selectors (continued)

- Statement sequences as clauses: 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
```

Selector Expressions

- In ML, F#, and LISP, the selector is an expression
- F#

```
let y =
   if x > 0 then x
   else 2 * x
```

  - If the `if` expression returns a value, there must be an `else` clause (the expression could produce output, rather than a value)

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. How are case values specified?
  5. What is done about unrepresented expression values?

Multiple-Way Selection: Examples

- C, C++, Java, and JavaScript

```
switch (expression) {
   case const_expr1: stmt1;
         …
   case const_exprn: stmtn;
   [default: stmtn+1]
}
```

Multiple-Way Selection: Examples

- 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: Examples

- 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: Examples

- Ruby has two forms of case statements - we'll cover only one

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

Implementing Multiple Selectors

- Approaches:
  - Multiple conditional branches
  - Store case values in a table and use a linear search of the table
  - When there are more than ten cases, a hash table of case values can be used
  - If the number of cases is small and more than half of the whole range of case values are represented, an array whose indices are the case values and whose values are the case labels can be used

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
```

Scheme's Multiple Selector

- General form of a call to **`cond`**:

```
(cond
   (predicate1 expression1)
   …
   (predicaten expressionn)
   [(ELSE expressionn+1)]
)
```

Scheme's Multiple Selector

- The **`else`** clause is optional; **`else`** is a synonym for true
- Each predicate-expression pair is a parameter
- Semantics: The value of the evaluation of **`cond`** is the value of the expression associated with the first predicate expression that is true

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?

Counter-Controlled Loops

- A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values
- 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?

Counter-Controlled Loops: Examples

- Ada

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

Counter-Controlled Loops: Ada

- 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

Counter-Controlled Loops: Examples

- C-based languages

```
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
  - If the second expression is absent, it is an infinite loop

Counter-Controlled Loops: C-based Languages

- Design choices:
  - 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
  - It is legal to branch into the body of a for loop in C

Counter-Controlled Loops: 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

Counter-Controlled Loops: Examples

- Python

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

Counter-Controlled Loops: Examples

- Python
  - 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
  - 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

Counter-Controlled Loops: Example in F#

- Because counters require variables, and functional languages do not have variables, counter-controlled loops must be simulated with recursive functions

```
let rec forLoop loopBody reps =
if reps <= 0 then ()
else
         loopBody()
         forLoop loopBody, (reps - 1)
```

- This defines the recursive function **`forLoop`** with the parameters **`loopBody`** (a function that defines the loop's body) and the number of repetitions
- () means do nothing and return nothing

Logically-Controlled Loops

- Repetition control is based on a Boolean expression
- Design issues:
  - Pretest or posttest?
  - Should the logically controlled loop be a special case of the counting loop statement or a separate statement?

Logically-Controlled Loops: Examples

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

```
while (control_expr)
     loop_body
```

or

```
do   {
     loop body
while (control_expr)
```

Logically-Controlled Loops: Examples

- In both C and C++ it is legal to branch into the body of a logically-controlled loop
- 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

Logically-Controlled Loops: Examples in F#

- As with counter-controlled loops, logically-controlled loops can be simulated with recursive functions

```
let rec whileLoop test body =
  if test() then
    body()
    whileLoop test body
  else ()
```

- This defines the recursive function whileLoop with parameters test and body, both functions. test defines the control expression

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
  - Should the conditional be part of the exit?
  - Should control be transferable out of more than one loop?

User-Located Loop Control Mechanisms

- C , C++, Python, Ruby, and C# have unconditional unlabeled exits (**`break`**)
- Java and Perl have unconditional labeled exits (**`break`** in Java, **`last`** in Perl)
- C, C++, and Python have an unlabeled control statement, **`continue`**, that skips the remainder of the current iteration, but does not exit the loop
- Java and Perl have labeled versions of **`continue`**

Iteration Based on Data Structures

- The number of elements in a data structure controls 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)){
 ...
}
```

Iteration Based on Data Structures (continued)

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

Iteration Based on Data Structures (continued)

- Java 5.0 (uses **`for`**, although it is called **`foreach`**)
- For arrays and any other class that implements the **`Iterable`** interface, e.g., **`ArrayList`**

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

Iteration Based on Data Structures (continued)

- C# and F# (and the other .NET languages) have generic library classes, like Java 5.0 (for arrays, lists, stacks, and queues). Can iterate over these with the **`foreach`** statement. User-defined collections can implement the **`IEnumerator`** interface and also use **`foreach`**.

Iteration Based on Data Structures
(continued)

```
List<String> names = new List<String>();

names.Add("Bob");
names.Add("Carol");
names.Add("Ted");

foreach (Strings name in names)
     Console.WriteLine ("Name: {0}",
                    name);
```

Iteration Based on Data Structures in Ruby

- Ruby blocks are sequences of code, delimited by either braces or do and end
- Blocks can be used with methods to create iterators

Iteration Based on Data Structures in Ruby (continued)

- Predefined iterator methods (**`times`**, **`each`**, **`upto`**):

```
3.times {puts "Hey!"}
list.each {|value| puts value}
(list is an array; value is a block parameter)
1.upto(5) {|x| print x, " "}
```

- Ruby has a for statement, but Ruby converts them to **`upto`** method calls

Iteration Based on Data Structures in Ada

- Ada allows the range of a loop iterator and the subscript range of an array be connected

```
subtype MyRange is Integer range 0.99;
MyArray: array (MyRange) of Integer;
for Index in MyRange loop
  ...MyArray(Index) ...
end loop;
```

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

Guarded Commands

- Designed 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 expr> -> <statement>
[] <Boolean expr> -> <statement>
 ...
[] <Boolean expr> -> <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

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

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