Organization of Programming Languages
CS3200 / 5200N

Lecture 08

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Control Flow

- **Control flow** = the flow of control, or execution sequence, in a program.

- Levels of control flow:
  1. Within **expressions**.
  2. Among **program statements**.
  3. Among **program units**.

Structured Control Flow

- A program is called **structured** if the flow of control is evident from the syntactic/static structure of the program.
- **Structured programming** allows the programmer to be able to reason about the behaviour of a program by just analyzing the program text:
  - Eliminates some of the complexity that arises when programs become large.
  - Common patterns of control flow that are used over and over by the programmers are integrated in special control statements in the language:
    - **selection** statements.
    - **iteration** statements.

Selection Statements

- A **selection statement** provides the means of choosing between two or more paths of execution.

- Two general categories:
  - Two-way selectors (***if-then-else***)
  - Multiple-way selectors (***switch*** *or* ***case***).

Two-Way Selection Statements

- General form:

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

- Nested selectors: which if is paired with the else?

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

Nested Selectors

- Static semantics rule (C/C++/Java/C#):
  - `else` matches with the nearest `if`.
- To force an alternative semantics, compound statements may be used:

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

- Perl requires that all then & else clauses to be compound.

Nested Selectors

- Statement sequences as clauses: Ruby

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

Nesting Selectors

- Statement sequences as clauses: Python

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

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups.
- C/C++/Java:

```
switch (expression) {
  case const_expr_1: stmt_1;
  …
  case const_expr_n: stmt_n;
  [default: stmt_n+1]
}
```

- C# disallows the implicit execution of more than one segment (need explicit *break* or *goto*).

Multiple-Way Selection Statements: C/C++/ Java

- Control is allowed to fall through more than one segment:

```
switch (index) {
  case 1:
  case 3: odd++;
          break;
  case 2:
  case 4: even++;
          break;
  default: cout << "Unknown index " << index;
}
```

Multiple-Way Selection Statements: C#

- Need explicit transfer control through *break* or *goto*:

```
switch (value) {
  case -1: negatives++;
           break;
  case 0: zeros++;
          goto case 1;
  case 1: positives++;
          break;
  default: Console.WriteLine("Unexpected value");
}
```

- Control and case expressions can also be strings.

Multiple-Way Selection Statements: C/C++/ Java

- No restriction on placement of case expressions in C/C++:

```
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);
```

- Case expressions allowed to appear only at top level in Java.

Multiple-Way Selection Statements: Ada

- Ada's `case` is more reliable than C's `switch`:
  - once a segment execution is completed, control is passed to the first statement after the `case` statement.
  - choice lists need to be exhaustive.
- Can use subranges 10 .. 20, or disjunctions 10 | 15 | 20.

```
case expression is
 when choice_list => stmt_sequence;
 …
 when choice_list => stmt_sequence;
 [when others => stmt_sequence;]
end case;
```

Multiple-Way Selection Using `else-if`

- Multiple-Way selectors can appear as direct extensions to Two-Way selectors, using else-if clauses.

- Python:

```
if count < 10:
  bag1 = True
elif count < 100:
  bag2 = True
elif count < 1000:
  bag3 = True
else:
  bag4 = True
```

- Ruby:

```
case
  when count < 10 then bag1 = True
  when count < 100 then bag2 = True
  when count < 1000 then bag3 = True
  else bag4 = True
end
```

Multiple-Way Selection Statements: Ruby

- Case constructs are expressions:

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

Iteration Statements

- The repeated execution of a statement or compound statement can be accomplished by:
  - **iteration** (imperative languages).
  - **recursion** (functional languages).

- Iteration Statements provide for structured iteration without the use of goto statements:
  - **Counter-Controlled** Loops (*definite* iterations).
  - **Logically-Controlled** Loops (*indefinite* iterations).

Definite vs. Indefinite Iterations

- A *definite* iteration is executed a fixed number of times:

```
for (int i = 0; i < 10; i++) {
    sum = sum + a[i];
}
```

- An *indefinite* iteration relies on a dynamically computed value to determine whether the iteration should continue:

```
int fact = 1;
while (n > 1) {
    fact = fact * n;
    n = n - 1;
}
```

Common Iteration Constructs in C/C++/Java

- **while loops** (pretest):

```
while (<condition>) <statement>;
while (<condition>) {<statement>; <statement>; …}
```

- **do-while loops** (posttest, similar to **repeat-until** in Pascal):

```
do <statement> while (<condition>);
do {<statement>; <statement>; …} while (<condition>);
```

- **for loops** (restricted form of while loops):

```
for (<initialize>; <test>; <step>) <statement>
for (<initialize>; <test>; <step>) {<statement-list>}
```

- Exercise:
  - state semantics for each construct (natural language, denotational).
  - model for loops using while loops.

Iteration Constructs in Ada

- **for loops**:

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

- Ada vs. C differences:
  - The loop variable does not exist outside the loop. and cannot be changed in the loop.
  - The discrete range is evaluated just once.
  - Cannot branch into the loop body.

```
Count: Float := 3.14;
for Count in 1..10 loop
   Sum := Sum + Count;
end loop;
```

Iteration Constructs in Python

- **for loops**:

```
for <var> in <domain>:
  <loop-body>
[else:
  <else-clause>]
```

- The domain is often a range:
  - a list of values in brackets ([2, 4, 6]);
  - a call to the range function, e.g. range(4) which returns [0, 1, 2, 3]

- The else clause is optional, and is executed if the loop terminates normally.

Special Iteration Constructs: `break`

- Most of the time iteration constructs are *single-entry*, *single-exits*.

- Sometimes a loop needs to be terminated prematurely, if a special condition arrives:
  - C /C++/C#, Python, and Ruby have unconditional unlabeled exits (`break`):
    - transfer control right after the end of the enclosing loop.
  - Java and Perl have unconditional labeled exits (`break` in Java, `last` in Perl):
    - transfer control at the labeled statement..

Special Iteration Constructs: `continue`

- Sometimes it is necessary to force a loop to be re-entered from the "top" before the loop has reached the "bottom":
  - C/C++ and Python have an unlabeled control statement (`continue`).
  - Java and Perl have labeled versions of `continue`.

```
outerloop: // Java
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    if (a[i][j] < 0)
      break outerloop;
  }
}
```

Iteration Statements

- Iteration constructs, along with `break` and `continue` are just a more structured way of programming common `goto` control flow.
- For example, the while loop:

```
start: // start of the loop
  if (cond-expr == false)
     goto end;
  ... // body of the loop
  goto start;
end: // end of the loop
... // statements following the loop
```

Reading Assignment

Chapter 8 (8.1 – 8.4)

Special Iteration Constructs

- Infinite loops:

```
while (true) { ... };
for (;;) { ... };
```

- Execute-once loops:

```
do { ... } while (false);
```