Programming Languages

Tevfik Koşar

Lecture - XI
February 21st, 2006

Roadmap

- Control Flow
  - Basic Paradigms
- Expression Evaluation
Control Flow or Ordering

- Control flow/ordering is fundamental in most models of computing
- Basic paradigms for control flow/ordering:
  - Sequencing: the order in which statements appear in code
  - Selection: making choices depending on a run-time condition
  - Iteration: executing a fragment of code repeatedly
  - Procedural abstraction: subroutines, functions
  - Recursion: Defining an expression in terms of itself
  - Concurrency: Execution of two or more program fragments at the same time
  - Nondeterminacy: the ordering is unspecified

Expression Evaluation

- Different styles:
  - Infix: \((a + b) \times c\)
  - Prefix: \(^{\times}^{(\times a b) c}\) --eg. Lisp
- In Ada, \(a+b\) is short for \(^{\times}(a, b)\);
- In C++, \(a+b\) is short for \(a.aoperator+b\);
- Precedence, associativity (see Figure 6.1)
  - C has 15 levels - too many to remember
  - Pascal has 4 levels - too few for good semantics
  - Fortran has 8
  - Ada has 6
    - Ada puts and \& or at same level
  - Lesson: when unsure, use parentheses!
Expression Evaluation

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Fortran</th>
<th>Pascal</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>not</td>
<td>++, -- (post-inc., dec.)</td>
<td>abs (absolute value), not, ++</td>
<td></td>
</tr>
<tr>
<td>* /</td>
<td>* / div, mod</td>
<td>* / (binary), /, % (modulus division)</td>
<td>* /, mod, rem</td>
<td></td>
</tr>
<tr>
<td>+ -</td>
<td>+ - (binary)</td>
<td>+ - (binary)</td>
<td>+ - (binary)</td>
<td></td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>&lt; &gt; (less than, greater than)</td>
<td>&lt; &gt; (less than, greater than)</td>
<td>&lt; &gt; (less than, greater than)</td>
<td></td>
</tr>
<tr>
<td>.eq., .ne., .lt., .le., .ge., .gt.</td>
<td>.eq., .ne., .lt., .le., .ge., .gt.</td>
<td>!, ==, !=, &gt;=, &lt;=, &gt;, &lt; (equality tests)</td>
<td>!, /=, &lt;, &lt;=, &gt;, &gt;=</td>
<td></td>
</tr>
<tr>
<td>.and.</td>
<td>.and.</td>
<td>&amp;&amp; (logical and)</td>
<td>and, or, xor</td>
<td></td>
</tr>
<tr>
<td>.or.</td>
<td>.or.</td>
<td></td>
<td></td>
<td>(logical or)</td>
</tr>
<tr>
<td>.equiv., .nequiv.</td>
<td>.equiv., .nequiv.</td>
<td>?; (if-then-else)</td>
<td>(logical comparisons)</td>
<td></td>
</tr>
<tr>
<td>+, -, /, &lt;&lt;, &gt;&gt;, &amp;,</td>
<td>+, -, /, &lt;&lt;, &gt;&gt;, &amp;,</td>
<td>+=, -=, *=, /=, %=, &lt;&lt;=, &gt;&gt;=, &amp;=, ^=,</td>
<td>= (assignment)</td>
<td></td>
</tr>
<tr>
<td>, (sequencing)</td>
<td>, (sequencing)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: Operator precedence levels in Fortran, Pascal, C, and Ada. The operators at the top of the figure group most tightly.

Precedence:
- If A>B and C>D then...
  Causes an error in Pascal since it is grouped as:
  If A< (B and C) <D then...

Associativity:
- Basic arithmetic expressions almost always associate from left to right
  - 9-3-2 is 4 not 8
- In Fortran, exponentiation operator associates from right to left
  - 2**3**2 is 512 (2**9) not 64 (8**2)
- In Ada exponentiation does not associate
  - You have to write (2**3)**2 or 2**(3**2)
- In C, assignment associates from right to left
  - a=b=a+c;
Expression Evaluation

• **Short-circuiting**
  – Consider \((a < b) \&\& (b < c)\):
    • If \(a \geq b\) there is no point evaluating whether \(b < c\)
      because \((a < b) \&\& (b < c)\) is automatically false
  – Other similar situations
    1. if \((b \neq 0 \&\& a/b == c)\) ...
    2. if \((p \&\& p->foo)\) ...
    3. if \((f || messy())\) ...

Expression Evaluation

• **Orthogonality**
  – Features that can be used in any combination
    • Meaning is consistent
      1. if \((\text{if } b \neq 0 \text{ then } a/b == c \text{ else } \text{false})\) then ...
      2. if \((\text{if } f \text{ then } \text{true} \text{ else } \text{messy()}\) then ...
    • First example: Algol 68
      1. \(a := \text{begin } f(b); g(c) \text{ end;}

  • In C:
    1. If \((a = b())\) ...
    2. If \((a == b = c())\) ...
Expression Evaluation

Combination of Assignment Operators:

In C:

- \( a = a + 1; \)  \( \Rightarrow \) \( a++; \) or \( ++a; \)
- \( a = a + b; \)  \( \Rightarrow \) \( a += b; \)
- \( A[--i] = b; \) \( A[i--] = b; \) difference??

\( t1=p; p = p+1; \) \( t2=q; q = q+1; \) \( *t1 = *t2; \)
\( \Rightarrow \) \( *p++ = *q++; \)

Expression Evaluation

In languages like Clu, ML, Perl, Python, it is possible..

- **Multiway Assignment:**
  - \( a, b := c, d \)  \( \Rightarrow \) \( a:=c; b:=d; \)
  - \( a, b := b,a; \)  \( \Rightarrow \) \( temp:=a; a:=b; b:=temp; \)
Expression Evaluation

• Assignment
  – statement (or expression) executed for its side effect
  – assignment operators (+=, -=, etc)
    • handy
    • avoid redundant work (or need for optimization)
    • perform side effects exactly once
  – C --, ++
    • postfix form

Applying Mathematical Identities:
  a = b+c;
  d = c+e+b;
Will be rearranged by some compilers as:
  a = b+c;
  a = b+c+e;
Then, code is generated for:
  a = b+c;
  d = a+e;
Flow

- Unstructured Flow
  - Go To

- Structured Flow
  - While loop
  - For loop

Sequencing

- Sequencing
  - specifies a linear ordering on statements
    - one statement follows another
  - very imperative, Von-Neuman
Selection

- Selection
  - sequential if statements
    - \[ \text{if ... then ... else} \]
    - \[ \text{if ... then ... elsif ... else} \]
  - In Lisp: \[
    \text{(cond}
    \quad \text{(C1) (E1)}
    \quad \text{(C2) (E2)}
    \quad \vdots
    \text{)}
\]
- Case/switch statements (eg. Pascal/C)
  - Case ...
    - 1: clause_A;
    - 2,3: clause_B;
    - End;

Selection

- Code generated w/o short-circuiting (Pascal)
  - \( r1 := A \) -- load
  - \( r2 := B \)
  - \( r1 := r1 > r2 \)
  - \( r2 := C \)
  - \( r3 := D \)
  - \( r2 := r2 > r3 \)
  - \( r1 := r1 \& r2 \)
  - \( r2 := E \)
  - \( r3 := F \)
  - \( r2 := r2 <> r3 \)
  - \( r1 := r1 \\ r2 \)
  - if \( r1 = 0 \) goto L2
  - L1: then_clause -- label not actually used
  - goto L3
  - L2: else_clause
  - L3:
Selection

- Code generated w/ short-circuiting (C)

```
r1 := A
r2 := B
if r1 <= r2 goto L4
r1 := C
r2 := D
if r1 > r2 goto L1
L4:    r1 := E
        r2 := F
        if r1 = r2 goto L2
L1:    then_clause
        goto L3
L2:    else_clause
L3:
```