

Programming Languages Lecture09 – Control Flow



남 범 석 bnam@skku.edu



- Basic paradigms for control flow
 - Sequencing
 - Selection 📃
 - Iteration
 - Procedural Abstraction
 - Recursion
 - Concurrency
 - Exception handling and speculation
 - Non-determinacy

Expression Evaluation

- Infix, prefix operators
 - Prefix notation does not incur ambiguity
 - Infix notation leads to ambiguity without parentheses
- Precedence, associativity
 - C has 15 levels too many to remember
 - Pascal has 3 levels too few for good semantics
 - Fortran has 8 levels
 - Ada has 6 levels
- Lesson
 - When unsure, use parentheses!

Precedence for Infix Notations

Fortran	Pascal	C	Ada
		++, (post-inc., dec.)	
**	not	++, (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value) not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*,/,mod,rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /= , <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		(bit-wise inclusive or)	
and.		&& (logical and)	and, or, xor (logical operators)
.or.		(logical or)	
.eqv., .neqv. (logical comparisons)		?: (ifthenelse)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, = (assignment)	
		, (sequencing)	

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

Short-Circuiting

• Q:What will be printed?

```
int main()
{
   int x=0;
   if(0 && (x=1)==1) {
        x++;
   }
   printf("x=%d\n", d);
}
```

Short-Circuiting

- Evaluating partial boolean expressions, not all
 - Consider (a < b) && (b < c)
 - If a >= b there is no point evaluating whether b < c</p>
 - because (a < b) && (b < c) is automatically false
 - Other similar situations

```
if (b != 0 && a/b == c) ...
if (*p && p->foo) ...
if (f || messy()) ...
```

Short-circuiting improves the performance

What if short-circuiting is not provided as in Pascal?

Value vs. Location

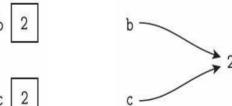
Assignment statement

- Value model
 - Expression can be either I-value or r-value
 - Not all expressions can be an I-value (e.g. 2+3 = a)
- Reference model
 - Every variable is an I-value
 - When a variable is used for r-value, it must be dereferenced to obtain the value. (It is done implicitly and automatically.)

Value Model vs. Reference Model

- Variables as values vs. variables as references
 - Value-oriented languages
 - C, Pascal, Ada
 - Reference-oriented languages
 - Most functional languages (Lisp, Scheme, ML)
 - Clu, Smalltalk
 - Java deliberately in-between 📁 💆
 - Built-in (primitive) types are valuese.g. byte, char, int, float, boolean, ...
 - User-defined types are references to objects
 e.g. all classes including Integer,
 Double, String, etc

4 a →



Value model Reference model

Expression- vs. Statement-Oriented

- Expression-oriented languages:
 - No separate notion of expression and statement
 - Functional languages (Lisp, Scheme, ML), Algol-68

Algol68 a := if b < c then d else e; a := begin f(b), g(c) end g(d); 2 + 3;</pre>

Statements in other languages can be used as expressions (the value of last statement is used)

Expressions are used and the results are thrown away

- Statement-oriented languages:
 - Most imperative languages
- C is kind a halfway in-between
 - Allows expression to appear instead of statement

```
if (a==b) { ... }
if (a=b) { ... }
```

Side Effects

- Side effect is a permanent state change by a function
 - Often discussed in the context of functions
 - Some noticeable effect of call other than return value
- In a more general sense, assignment statements provide the ultimate example of side effects
 - They change the value of a variable
 - Side effects are fundamental to the whole von Neumann model of computing
 - In (pure) functional, logic, and dataflow languages, there are no such changes (single-assignment languages)
 - But side effect can be nice for some functions e.g. rand()

Sequencing

- Sequencing
 - Specifies a linear ordering on statements
 - One statement follows another

```
E.g.) beginstatement I;statement 2;statement 3;end
```

Selection

Selection

• Same meaning as series of if-then-else statements

```
if ... then ...
else if ... then ...
else if ... then ...
else if ...
```

Examples

<Modula-2 >

```
IF a = b THEN ...
ELSIF a = c THEN ...
ELSIF a = d THEN ...
ELSE ...
END
```

```
<Lisp>
```

```
(cond

((= A B) (Expr<sub>1</sub>))

((= A C) (Expr<sub>2</sub>))

((= A D) (Expr<sub>3</sub>))

(T (Expr<sub>t</sub>))
```

Selection Implementation

- Conditional branch instruction
 - For simple selections
- Jump code
 - For general selections and logically-controlled loops
 - Especially useful in the presence of short-circuiting

```
if ((A > B) and (C > D)) or (E <> F) then
    then_clause
else
    else_clause
```

Code for No Short-Circuiting

Code generated w/o short-circuiting (Pascal)

```
□ r1 := A
                           ((A > B) \text{ and } (C > D)) \text{ or } (E <> F)
        r2 := B
        r1 := r1 > r2
        r2 := C
        r3 := D
        r2 := r2 > r3
        r1 := r1 \& r2
        r2 := E
        r3 := F
        r2 := r2 \neq r3
        r1 := r1 | r2
        if r1 = 0 goto L2
L1:
       then clause
                                 -- label not actually used
        goto L3
L2:
        else clause
L3:
```

Code for Short-Circuiting

Code generated w/ short-circuiting (C)

```
((A > B) \text{ and } (C > D)) \text{ or } (E <> F)
        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
      then clause
L1:
       goto L3
L2:
      else_clause
                                 No need to compute the whole boolean
L3:
                                 value into a register and test it for
                                 conditional jump
```

Selection - case/switch

Sequence of if-then-else (nested if-then-else) can be rewritten as case/switch

<Modula-2> CASE ... OF 1: clause_A | 2, 4: clause_B | 3, 6: clause_C ELSE clause_D END

Jump Tables for case/switch

- (Linear) jump tables
 - Instead of sequential test, compute address to jump to

```
&L1
T:
                             -- case 1
    &L2
                            -- case 2
                                            Jump table
    &L3
                            -- case 3
    &L2
                            -- case 4
    &L4
                            -- case 5
    &L3
                            -- case 6
L5: r1 := ...
                            -- calculate tested expr
    if r1 < 0 or r1 > 6 goto L4 -- ELSE case
    r2 := T[r1-1]
                           -- subtract off lower bound
    qoto *r2
L6:
L1: clause A
    goto L7
L2: clause B
    goto L7
```

Alternative Implementations

- Linear jump table is fast for case/switch
 - Also space efficient, if overall set of cases are dense and does not contain a large ranges
 - May consume extraordinary space for large value ranges

Alternatives

- Sequential testing (nested ifs), O(n)
 - Good for small number of cases
- Hashing, O(1)
 - Attractive for large label values
 - But space inefficient for large value ranges
- Binary search, O(logn)
 - Accommodate ranges easily

Iteration

- Logically-controlled loops
 - Controlled by a boolean expression

```
While condition_expr do .... enddo
```

- Enumeration-controlled loops
 - i: index of the loop, loop variable
 - Controlled by index's initial value, its bound, and step size
 - Semantic complications
 - Loop enter/exit in other ways
 - E.g., break, continue
 - Scope of control variable
 - Changes to bounds within loop
 - Changes to loop variable within loop
 - Value after the loop

```
do i = 1, 10, 2
...
enddo
```



- Recursion is equally powerful to iteration
 - Often more intuitive (sometimes less)
 - Naïve implementation is less efficient
 - If a recursive call is actually implemented with a subroutine call, it will allocate space for its function frame (local variables, bookkeeping information)
 - Compiler optimizations is required to generate excellent code for recursion (e.g., tail recursion)

Tail Recursion

Tail recursion

- A special kind of recursion where the recursive call is the very last thing in the function.
- I.e., it's a function that does not do anything at all after recursion.
- l.e., if no computation follows a recursive call, we call it tail recursion

■ Tail recursion is desirable for optimization

- Compiler can optimize it easily
- The information to store for the previous function is only the return address, since nothing to compute after return
- Otherwise, we may need to keep local variables for the remaining computation after recursive calls

Tail Recursion Elimination

```
int gcd (int a, int b) { /* assume a, b > 0 */
   if (a == b) return a;
   else if (a > b) return gcd (a - b,b);
   else return gcd (a, b - a);
}
```



Return multiple times at the end

```
int gcd (int a, int b) { /* assume a, b > 0 */
start:
   if (a == b) return a;
   else if (a > b) { a = a - b; goto start; }
   else { b = b - a; goto start; }
}
```

Return once at the end

Summary

- Expression
 - Evaluation order commutativity, associativity
- Control flow
 - Sequencing
 - Selection
 - Short-circuiting, jump table, ...
 - Iteration (loop)
 - Logically-controlled, enumeration-controlled
 - Recursion
 - Optimization for tail-recursion