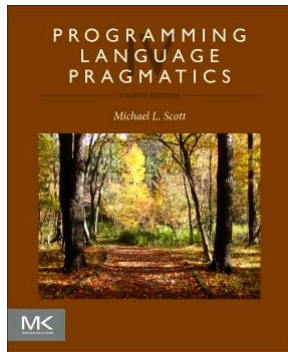


Chapter 6:: Control Flow

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Control Flow

- Basic paradigms for control flow:
 - Sequencing
 - Selection
 - Iteration
 - Procedural Abstraction
 - Recursion
 - Concurrency
 - Exception Handling and Speculation
 - Nondeterminacy



Expression Evaluation

- Infix, prefix operators
- Precedence, associativity (see Figure 6.1)
 - C has 15 levels - too many to remember
 - Pascal has 3 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

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)		?: (if...then...else)	
		=, +=, -=, *=, /=, %= >>=, <<=, &=, ^=, = (assignment)	
		, (sequencing)	

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



Expression Evaluation

- Ordering of operand evaluation (generally none)
- Application of arithmetic identities
 - distinguish between commutativity, and (assumed to be safe)
 - associativity (known to be dangerous)
 $(a + b) + c$ works if $a \sim \text{maxint}$ and $b \sim \text{minint}$ and $c < 0$
 $a + (b + c)$ does not
 - inviolability of parentheses

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

- if $(b \neq 0 \ \&\& \ a/b == c)$...

- if $(*p \ \&\& \ p \rightarrow \text{foo})$...

- if $(f \parallel \text{messy}())$...

- Can be avoided to allow for side effects in the condition functions



Expression Evaluation

- Variables as values vs. variables as references
 - value-oriented languages
 - C, Pascal, Ada
 - reference-oriented languages
 - most functional languages (Lisp, Scheme, ML)
 - Clu, Smalltalk
 - Algol-68 kinda halfway in-between
 - Java deliberately in-between
 - built-in types are values
 - user-defined types are objects - references



Expression Evaluation

- Expression-oriented vs. statement-oriented languages
 - expression-oriented:
 - functional languages (Lisp, Scheme, ML)
 - Algol-68
 - statement-oriented:
 - most imperative languages
 - C kinda halfway in-between (distinguishes)
 - allows expression to appear instead of statement

Expression Evaluation

- Orthogonality
 - Features that can be used in any combination
 - Meaning is consistent

```
if (if b != 0 then a/b == c else false) then ...
```

```
if (if f then true else messy()) then ...
```

- Initialization
 - Pascal has no initialization facility (assign)
- Aggregates
 - Compile-time constant values of user-defined composite types



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



Expression Evaluation

- Side Effects
 - often discussed in the context of functions
 - a side effect is some permanent state change caused by execution of function
 - 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



Expression Evaluation

- SIDE EFFECTS ARE FUNDAMENTAL TO THE WHOLE VON NEUMANN MODEL OF COMPUTING
- In (pure) functional, logic, and dataflow languages, there are no such changes
 - These languages are called SINGLE-ASSIGNMENT languages



Expression Evaluation

- Several languages outlaw side effects for functions
 - easier to prove things about programs
 - closer to mathematical intuition
 - easier to optimize
 - (often) easier to understand
- But side effects can be nice
 - consider `rand()`



Expression Evaluation

- Side effects are a particular problem if they affect state used in other parts of the expression in which a function call appears
 - It's nice not to specify an order, because it makes it easier to optimize
 - Fortran says it's OK to have side effects
 - they aren't allowed to change other parts of the expression containing the function call
 - Unfortunately, compilers can't check this completely, and most don't at all



Sequencing

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



Selection

- Selection

- sequential if statements

- if ... then ... else

- if ... then ... elsif ... else

- (cond

- (C1) (E1)

- (C2) (E2)

- ...

- (Cn) (En)

- (T) (Et)

-)



Selection

- Selection
 - Fortran computed gotos
 - jump code
 - for selection and logically-controlled loops
 - no point in computing a Boolean value into a register, then testing it
 - instead of passing register containing Boolean out of expression as a synthesized attribute, pass inherited attributes INTO expression indicating where to jump to if true, and where to jump to if false



Selection

- Jump is especially useful in the presence of short-circuiting
- **Example** (section 6.4.1 of book):

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

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:
```



Iteration

- Enumeration-controlled
 - Pascal or Fortran-style for loops
 - scope of control variable
 - changes to bounds within loop
 - changes to loop variable within loop
 - value after the loop
 - Can iterate over elements of any well-defined set



Recursion

- Recursion
 - equally powerful to iteration
 - mechanical transformations back and forth
 - often more intuitive (sometimes less)
 - *naïve* implementation less efficient
 - no special syntax required
 - fundamental to functional languages like Scheme



Recursion

- Tail recursion
 - No computation follows recursive call

```
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);  
}
```

