Chapter 6:: Control Flow

Programming Language Pragmatics

Michael L. Scott

ELSEVIER

Control Flow

- control flow or ordering
  - fundamental to most models of computing
  - determines ordering of tasks in a program

Control Flow

- basic categories for control flow
  - sequencing: order of execution
  - selection (also alternation): choice among two or more statements or expressions
    - **`if`** or **`case`** statements
  - iteration: loops
    - **`for, do, while, repeat`**
  - procedural abstraction: parameterized subroutines

Control Flow

- basic categories for control flow (cont.)
  - recursion: expression defined in terms of (simpler versions of) itself
  - concurrency: two or more program fragments are executed at the same time
    - in parallel on separate processors
    - interleaved on a single processor
  - exception handling and speculation
  - nondeterminancy: order or choice is deliberately left unspecified

Control Flow

- previous eight categories cover all of the control-flow constructs in most programming languages
- better to think in these categories rather than the specifics of a single programming language
  - easier to learn new languages
  - evaluate tradeoffs among languages
  - design and evaluate algorithms

Control Flow

- importance of different categories varies across programming language paradigms
  - sequencing central in imperative and object-oriented languages, but less important in functional languages
  - functional languages use recursion heavily, while imperative languages focus more on iteration
  - logic languages hide control flow entirely and allow the system to find an order in which to apply inference rules

Expression Evaluation

- expression consists of a simple object (literal, variable, constant) or an operator or function call
  - function: **`my_func(A, B, C)`**
  - operators: simple syntax, one or two operands
    - **`a + b`**
    - **`-c`**
  - sometimes operators are syntactic sugar
    - in C++, **`a + b`** short for **`a.operator+(b)`**
- some languages impose an ordering for operators and their operands
  - prefix and postfix sometimes referred to as Polish prefix and Polish postfix after Polish logicians who studied and popularized them

Prefix, Infix, and Postfix Notation

- ordering for operators and their operands
  - prefix: op a b or op(a,b)
    - Lisp: `(* (+ 1 3) 2)`
    - Cambridge prefix: function name inside parentheses; also used with multiple operands: `(+ 2 4 5 1)`
  - infix: a op b
    - standard method
    - C: `a = b != 0 ? a/b : 0`
  - postfix: a b op
    - least common - used in Postscript, Forth, and intermediate code of some compilers
    - C (and its descendants): `x++`
    - Pascal: pointer dereferencing operator (`^`)

Expression Evaluation

- arithmetic and logic operations may be ambiguous without parentheses
  - Fortran: **`a + b * c**d**e/f`**
- languages set precedence and associativity rules to determine order of operations
  - precedence rules: order of types of operations
    - **`2 + 3 * 4`** (**`14`** or **`20`**?)
  - associativity rules: order of operations at same precedence
    - **`9 – 3 – 2`** (**`4`** or **`8`**?)

Expression Evaluation

- languages have individual precedence and associativity rules
  - C has 15 levels – too many to remember
  - Pascal has 3 levels – too few for good semantics
  - Fortran has 8
  - Ada has 6
  - 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

- example:
  - `3 + 2**2**3`
    - exponentiation has higher precedence than addition
    - exponentiation has right to left associativity
  - use parentheses to force other interpretations
    - `3 + 2**(2**3)`
    - `(3 + 2)**2**3`

Assignment

- typically, a variable takes on a new value
- assignment is a side effect
  - something that influences later computation or output and is not a return value
  - C: assignment does yield a value
- l-value: term on left side of **=**
- r-value: term on right side of **=**

Assignment

- ordering of operand evaluation
  - generally none
- application of arithmetic identities
  - commutativity is assumed to be safe
  - associativity (known to be dangerous)

    `(a + b) + c`

      works if **`a ~= maxint`** and **`b ~= minint`** and **`c < 0`**

    `a + (b + c)`

      does not

Expression Evaluation

- short-circuiting
  - consider **`(a < b) && (b < c)`**
    - if **`a >= b`** there is no point evaluating whether **`b < c`** because **`(a < b) && (b < c)`** is automatically false
  - other similar situations

```
if (b != 0 && a/b == c) ...
if (p && p->foo) ...
if (unlikely_condition && expensive_fn())...
```

- be cautious - need to be sure that your second half is valid, or else coder could miss a runtime error without proper testing

Expression Evaluation

- variables as values vs. variables as references
  - value-oriented languages
    - C, Pascal, Ada
  - reference-oriented languages
    - most functional languages (Lisp, Scheme)
  - Java deliberately in-between
    - built-in types are values
    - user-defined types are objects - references

Expressions vs. Statements

- most languages distinguish between expressions and statements
  - expressions always produce a value, and may or may not have a side effect
    - Python: **`b + c`**
  - statements are executed solely for their side effects, and return no useful value
    - Python: **`mylist.sort()`**
- a construct has a side effect if it influences subsequent computation in some way (other than simply returning a value)

Expression Evaluation

- expression-oriented vs. statement-oriented languages
  - expression-oriented
    - functional languages (Lisp, Scheme, ML)
  - statement-oriented:
    - most imperative languages
  - C halfway in-between
    - allows expression to appear instead of statement, but not the reverse

Assignment Shortcuts

- assignment
  - statement (or expression) executed for its side effect
    - key to most programming languages you have seen so far
  - assignment operators
    - **`+=`**, **`-=`**, etc.
    - handy shortcuts
    - avoid redundant work
    - reduce programmer errors
    - perform side effects exactly once
      - example: **`A[index_fn(i)]++;`**
      - vs. **`A[index_fn(i)] = A[index_fn(i)] + 1;`**

Multiway Assignment

- some languages (including Python and Ruby) allow multiway assignment
  - example: **`a,b = c,d;`**
  - defines a tuple, equivalent to **`a = c; b = d;`**
- can simplify computation
  - **`a,b = b,a`** (no need for a temp variable)
  - **`a,b,c = foo(d,e,f)`** (allows a single return)

C: Assignments within Expressions

- combining expressions with assignments can have unfortunate side effects, depending on the language
  - C has no true boolean type (just uses int’s or their equivalents), and allows assignments within expressions
  - example

```
if (a = 0) {
   …
}
```

What does this do?

Expression Evaluation

- side effects are a fundamental aspect of the whole von Neumann model of computation.
  - what is the von Neumann architecture?
- in (pure) functional and logic languages, there are no such changes
  - single-assignment languages
  - very different

Expression Evaluation

- some 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()`**

More on Side Effects

- side effects are a particular problem if they affect state used in other parts of the expression in which a function call appears
  - example: `a - f(b) - c*d`
  - good not to specify an order, because it makes it easier to optimize
  - unfortunately, compilers can't check this completely, and most don't at all

Code Optimization

- most compilers attempt to optimize code
  - example: **`a = b + c`** then **`d = c + e + b`**
- evaluating part of each statement can speed up code
  - **`a = b / c / d`** then **`e = f / d / c`**
  - **`t = c * d`** and then **`a = b / t`** and **`e = f / t`**
- arithmetic overflow can really become a problem here
  - can be dependent on implementation and local setup
  - checking provides more work for compiler, so slower
  - with no checks, these can be hard to find

Sequencing

- sequencing
  - specifies a linear ordering of statements
    - one statement follows another
  - imperative, Von-Neuman
- in assembly, the only way to "jump" around is to use branch statements
- early programming languages (such as C) mimicked this using **`goto`**

The End of goto

- in 1968, Edsger Dijkstra wrote an article condemning the **`goto`** statement
- while hotly debated, **`goto`** statements have essentially disappeared from modern programming languages
- did not fit structured programming model
  - top down design
  - modularization of code
  - structured types
  - descriptive variables
  - iteration

Alternatives to goto

- getting rid of `goto` was actually fairly easy, since it was usually used in certain ways
  - `goto` to jump to end of current subroutine
    - use return instead
  - `goto` to escape from the middle of a loop
    - use exit or break instead
    - much harder if nesting is deep
  - `goto` to repeat sections of code
    - use loops instead

Case for goto

- several settings are very useful for `goto` statements
  - to end a procedure/loop early (for example, if target value is found)
    - use break or continue instead
  - problem: bookkeeping
    - breaking out of code might end a scope
      - need to call destructors, deallocate variables, etc.
    - adds overhead to stack control
      - must be support for unwinding the stack

Case for goto

- another example: exceptions
- `goto` was generally used as error handling, to exit a section of code without continuing
- modern languages generally throw and catch exceptions instead
  - adds overhead
  - but allows more graceful recovery

Sequencing

- blocks of code are executed in a sequence
- blocks are generally indicated by { … } or similar construct
- interesting note: without side effects, blocks are essentially useless
  - the value is just the last return
- in some languages, functions which return a value are not allowed to have a side effect at all
  - any function call will have the same value, no matter when it occurs
  - not always desirable, of course
    - `rand` function definitely should not return the same value every time!

Selection

- selection: introduced in Algol 60
  - sequential if statements

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

  - Lisp variant

```
(cond
        (C1) (E1)
        (C2) (E2)
        ...
        (Cn) (En)
        (T)  (Et)
)
```

Selection

- Algol 60 example

```
if a = b then PROC := 2
elsif a = c then PROC := 3
elsif a = d then PROC := 4
else PROC := 1
end;
```

- Lisp variant

```
(cond
    ((= A B)  (2))
    ((= A C)  (3))
    ((= A D)  (4))
    (T        (1))
)
```

Selection

- selection
  - Fortran computed **`goto`** statements
  - 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: suppose code is generated for the following

```
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:
```

Selection: case/switch

- the case/switch statement was introduced to simplify certain if-else situations
- useful when comparing the same integer to a large variety of possibilities

Selection: case/switch

- example

```
i := ... (* potentially complicated expression *)
IF i = 1 THEN
    clause_A
ELSIF i IN 2, 7 THEN
    clause_B
ELSIF i IN 3..5 THEN
    clause_C
ELSIF (i = 10) THEN
    clause_D
ELSE
    clause_E
END
```

can be re-written as

```
CASE ... (* potentially complicated expression *) OF
    1:      clause_A
|   2, 7:   clause_B
|   3..5:   clause_C
|   10:     clause_D
    ELSE    clause_E
END
```

Selection: case/switch

- labels and arms must be disjoint
- label type must be discrete
  - integer, character, enumeration, subrange
- case/switch statements enhance code aesthetics, but principal motivation is to generate efficient target code

Selection: case/switch

- case can be translated as

```
      r1 := ...                  -- calculate tested expression
      if r1 ≠ 1 goto L1
      clause_A
      goto L6
L1:   if r1 = 2 goto L2
      if r1 ≠ 7 goto L3
L2:   clause_B
      goto L6
L3:   if r1 < 3 goto L4
      if r1 > 5 goto L4
      clause_C
      goto L6
L4:   if r1 ≠ 10 goto L5
      clause_D
      goto L6
L5:   clause_E
L6:
```

```
CASE ... (* potentially complicated expression *) OF
    1:       clause_A
|   2, 7:    clause_B
|   3..5:    clause_C
|   10:      clause_D
    ELSE     clause_E
END
```

Selection: case/switch

- can use an array of jump addresses (jump table) instead

```
T:   &L1                  -- tested expression = 1
     &L2
     &L3
     &L3
     &L3
     &L5
     &L2
     &L5
     &L5
     &L4                  -- tested expression = 10
L6:  r1 := . . .          -- calculate tested expression
     if r1 < 1 goto L5
     if r1 > 10 goto L5      -- L5 is the "else" arm
     r1 −:= 1                -- subtract off lower bound
     r2 := T[r1]
     goto *r2
L7:
```

```
CASE ... (* potentially complicated expression *) OF
    1:      clause_A
|   2, 7:   clause_B
|   3..5:   clause_C
|   10:     clause_D
    ELSE    clause_E
END
```

Selection: case/switch

- jump tables can take a lot of space if case covers large ranges or values or non-dense
- alternative methods
  - sequential testing
    - useful if number of case statements is small
  - hashing
    - useful if range of label values is large, but with many missing values
  - binary search
    - good for large ranges

Selection: case/switch

- languages differ in
  - syntax
  - punctuation
  - label ranges
  - default clause
    - some languages: else
    - Ada: all values must be covered
  - handling of match failures
    - some languages will require program failure for unmatched value
    - C and C++: no effect

Selection: case/switch

- C/C++/Java switch

```
switch (... /* tested expression */) {
    case 1:   clause_A
              break;
    case 2:
    case 7:   clause_B
              break;
    case 3:

    case 4:
    case 5:   clause_C
              break;
    case 10:  clause_D
              break;
    default:  clause_E
              break;
}
```

```
CASE ... (* potentially complicated expression *) OF
    1:       clause_A
|   2, 7:    clause_B
|   3..5:    clause_C
|   10:      clause_D
    ELSE     clause_E
END
```

Selection: case/switch

- C/C++/Java switch
  - each value must have its own label; no ranges allowed
  - lists of labels not allowed, but empty arms that fall through OK
  - **`break`** required at end of each arm that terminates
  - fall-through can cause unintentional hard-to-find bugs
    - C# requires each non-empty arm to end with **`break`**, **`goto`**, **`continue`**, or **`return`**
  - fall-through convenient at times

```
letter_case = lower;
switch (c) {
    ...
    case 'A' :
        letter_case = upper;
        /* FALL THROUGH! */
    case 'a' :
        ...
        break;
    ...
}
```

Iteration

- ability to perform some set of operations repeatedly
  - loops
  - recursion
- without iteration, all code would run in linear time
- most powerful component of programming
- in general, loops are more common in imperative languages, while recursion is more common in functional languages
  - loops generally executed for their side effects

Iteration

- enumeration-controlled loop
  - Pascal or Fortran-style for loops

```
do i = 1, 10, 2      -- index i, init val, bound, step
   …                 -- body will execute 5 times
enddo
```

  - changed to standard for loops later

```
FOR i := first TO last BY step DO
       …
END
```

Iteration: Code Generation

- none of these initial loops allow anything other than enumeration over a preset, fixed number of values
- results in efficient code generation

```
        R1 := first
        R2 := step
        R3 := last
        goto L2
    L1: …                   --loop body, use R1 for i
        R1 := R1 + R2
    L2: if R1 <= R3 goto L1
```

Iteration: Code Generation

- translation can be optimized if the number of iterations can be precomputed, although need to be careful of overflow
  - precompute total count, and subtract 1 each time until we hit 0
  - we must be able to precompute
    - always possible in Fortran or Ada, but C (and its descendants) are quite different

Iteration: Some Issues

- can control enter or leave the loop other than through enumeration mechanism?
  - **`break`**, **`continue`**, **`exit`**
  - Fortran allowed **`goto`** to jump inside a loop
- what happens if the loop body alters variables used to compute end-of-loop condition?
  - some languages only compute the bound once (not C)
- what happens if the loop modifies the index variable itself?
  - most languages prohibit this entirely, although some leave it up to the programmer
- can the program read the index after the loop has been completed, and if so, what is its value?
  - ties into issue of scope, and is very language-dependent

Iteration: Some Issues

- example: what happens if the loop modifies the index variable itself?

```
for i := 1 to 10 by 2
    …
    if i = 3
        i = 6
```

- example: can the program read the index after the loop has been completed, and if so, what is its value?

```
var c : 'a'..'z';
...
for c := 'a' to 'z' do begin
    ...
end;
(* what comes after 'z'? *)
```

Iteration: Combination Loops

- the **`for`** loop in C is called a combination loop
  - allows one to use more complex structures in the **`for`** loop
- the Modula-2 loop

```
FOR i := first TO last BY step DO
    ...
END
```

becomes

```
for (i = first; i <= last; i += step) {
    ...
}
```

which is equivalent to

```
i = first;
while (i <= last) {
    ...
    i += step;
}
```

Iteration: Combination Loops

- for loop useful in its compactness of clarity over while loop
- convenient to make loop iterator local to body of loop

```
for (int i = first; i <= last; i += step)
```

- essentially, for loops are another variant of while loops, with more complex updates and true/false evaluations each time
- operator overloading (such as operator++) combined with iterators actually allow highly non-enumerative for loops
- example

```
for (list<int>::iterator it = mylist.begin();
     it != mylist.end(); it++) {
   …
}
```

Iteration: Iterators

- languages such as Python and C# require any container to provide an iterator that enumerates items in that class
- example

```
for item in mylist:
  #code to look at items
```

Iteration: Logically Controlled Loops

- `while` loops are different from standard `for` loops
  - no set number of enumerations is predefined
- inherently strong
  - closer to `if` statements in some ways, but with repetition built in
- more difficult to code properly
- more difficult to debug
- code optimization is also harder
  - none of the `for` loop tricks will work

Recursion

- recursion
  - equally powerful to iteration
  - often more intuitive (sometimes less)
  - naive implementation less efficient
    - no special syntax required
    - fundamental to functional languages like Scheme

Recursion

- many criticize that recursion is slower and less efficient than iteration
  - alters the stack when calling a function
- a bit inaccurate – naively written iteration is probably more efficient than naively written recursion
- in particular, if the recursion is *tail recursion*, the execution on the stack for the recursive call will occupy the exact same spot as the previous method

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);
 }
```

  - a good compiler will translate this to machine code that runs in place
    - essentially returning to the start of the function with new **a,b** values

ELSEVIER

Recursion

- even if not initially tail recursive, simple transformations can often produce tail-recursive code
- additionally, clever tricks - such as computing Fibonacci numbers in an increasing fashion, rather than via two recursive calls - can make recursion comparable

Order of Evaluation

- generally, we assume that arguments are evaluated before passing to a subroutine, in applicative order evaluations
- not always the case: lazy evaluation or normal order evaluation pass unevaluated arguments to functions, and value is only computed if and when it is necessary
- applicative order is preferable for clarity and efficiency, but sometimes normal order can lead to faster code or code that won't give as many run-time errors
- in particular, for list-type structures in functional languages, this lazy evaluation can be key