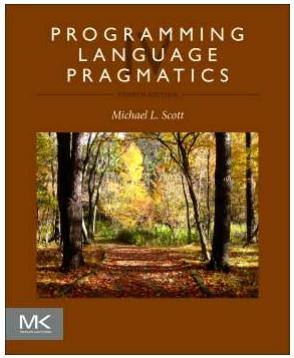
Chapter 6:: Control Flow

- CSE307/526: Principles of Programming Languages
- https://ppawar.github.io/CSE307-F18/index.html

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



ELSEVIER

Language Mechanisms for Control Flow

- Control flow/ordering in program execution:
  - –Ordering determines what should be done first, what second and so forth to accomplish desired task
- Successful programmer thinks in terms of basic principles of control flow, not in terms of syntax!

Control Flow

- Basic paradigms for control flow:
  - –Sequencing
  - –Selection - e.g. if and case(switch)
  - –Iteration - e.g. for, while loops
  - –Procedural abstraction – e.g. functions (Chapter 8)
  - –Recursion
  - –Concurrency (Chapter 12)
  - –Exception handling and speculation (roll back) (Chapter 8 and 12)
  - –Nondeterminacy (ordering or choice among statements is left unspecified)

Expression Evaluation

- Expression
  - –Simple object (e.g. constant)
  - –An operator or function applied to collection of operands or arguments
  - –Function call
- Notations

prefix:	*op* a b	or	*op*(a, b)	or	(*op* a b)	
infix:	a *op* b					
postfix:	a b *op*					

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

- Order of evaluation may influence result of computation.
- Purely functional languages:
  - –Computation is expression evaluation.
  - –The only effect of evaluation is the returned value.
- Imperative languages:
  - –Computation is a series of changes to the values of variables in memory.
  - –This is “computation by side effect”.
  - –The order in which these side effects happen may determine the outcome of the computation.

Assignments

- A programming language construct is said to have a side effect if it influences subsequent computation (and ultimately program output)
- Assignment is the simplest (and most fundamental) type of side effect a computation can have.
- Syntactic differences – semantically irrelevant
  - –A = 3 FORTRAN, PL/1, SNOBOL4, C, C++, Java
  - –A :- 3 Pascal, Ada, Icon, ML, Modula-3, ALGOL 68
  - –A <- 3 Smalltalk, Mesa, APL
  - –3 -> A BETA
  - –MOVE 3 TO A COBOL
  - –(SETQ A 3) LISP

Multiway Assignments

- In ML, Perl, Python, and Ruby:

```
a, b = c, d;
```

- Tuples consisting of multiple l-values and r-values
- The effect is: `a = c;` `b = d;`
- The comma operator on the left-hand side produces a tuple of l-values, while to comma operator on the right hand side produces a tuple of r-values.
- The multiway (tuple) assignment allows us to write things like: `a, b = b, a;` # that swap a and b

which would otherwise require auxiliary variables.

- Multiway assignment also allows functions to return tuples:

```
a, b, c = foo(d);  # foo returns a tuple of 3 elements
```

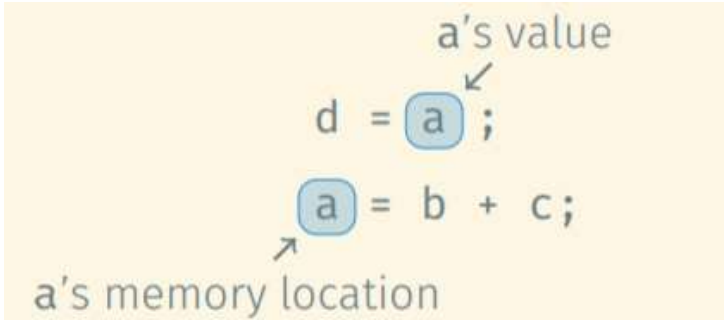
Definite Assignment

- the fact that variables used as r-values are initialized can be statically checked by the compiler.
  - –Every possible control path to an expression must assign a value to every variable in that expression!

```
int i;
int j = 3;
...
if (j > 0) {
    i = 2;
}
...                              // no assignments to j in here
if (j > 0) {
    System.out.println(i); // error: "i might not have been initialized"
}
```

References and values

- Expressions that denote values are referred to as r-values.
- Expressions that denote memory locations are referred to as l-values.
- In most languages (e.g. C), the meaning of a variable name differs depending on the side of an assignment statement it appears on:
  - –On the right-hand side, it refers to the variable's value—it is used as an r-value.
  - –On the left-hand side, it refers to the variable's location in memory—it is used as an l-value.



Variable models

- Value model
  - Assignment copies the value
- Reference model
  - A variable is always a reference
  - Assignment makes both variables refer to the same memory location
- Distinguish between:
  - Variables referring to the same object; and
  - Variables referring to different but identical objects

Value model vs. reference model

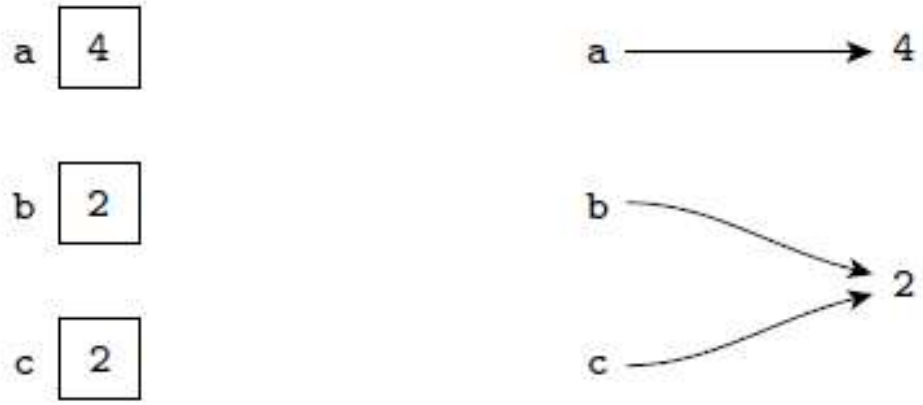


Figure 6.2 The value (left) and reference (right) models of variables. Under the reference model, it becomes important to distinguish between variables that refer to the same object and variables that refer to different objects whose values happen (at the moment) to be equal.

Variable models

- 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 is deliberately in-between
    - •built-in types are values
    - •user-defined types are objects - references

Java examples

```
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);
```

Output:

```
String a = "hi ";
String b = a;
b += "world";
System.out.println("a = " + a);
System.out.println("b = " + b);
```

Output:

```
Obj a = new Obj();
Obj b = a;
b.change();
System.out.println(a == b);
```

Output:

```
StringBuffer a = new StringBuffer();
StringBuffer b = a;
b.append("This is b's value.");
System.out.println("a = " + a);
System.out.println("b = " + b);
```

Output:

Java examples

```
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);
```

Output:

```
a = 5
b = 15
```

```
Obj a = new Obj();
Obj b = a;
b.change();
System.out.println(a == b);
```

Output:

```
true
```

```
String a = "hi ";
String b = a;
b += "world";
System.out.println("a = " + a);
System.out.println("b = " + b);
```

Output:

```
a = hi
b = hi world
```

```
StringBuffer a = new StringBuffer();
StringBuffer b = a;
b.append("This is b's value.");
System.out.println("a = " + a);
System.out.println("b = " + b);
```

Output:

```
a = This is b's value
b = This is b's value
```

Associativity rules, execution ordering

- Associativity rules specify whether sequences of operators of equal precedence group to the right or to the left
  - –Subtraction associates left-to-right. 9 – 3 – 2 = 4 and not 8.
  - –Exponentiation operator (**) associates right-to-left. 4**(3**2) = 262,144 and not (4**3)**2 = 4,096.
- Execution ordering is not necessarily defined
  - –In (1<2 and 3>4), which is evaluated first?
  - –Some languages define order left to right
  - –Some allow re-order

An example in C

```
for(i = m = M = 1; N - ++i; M = m + (m = M));
```

- What does this code compute?
- The answer depends on the evaluation order of two subexpressions of M = m + (m = M)

Probably intended

N	m	M
2	1	1
3	1	2
4	2	3
5	3	5
6	5	8

$M = F_N$ (Nth Fibonacci number)

Actual

N	m	M
2	1	1
3	1	2
4	2	4
5	4	8
6	8	16

$M = 2^{N-2}$

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
  - (and a b): If a is false, b has no effect on the value of the whole expression.
  - (or a b): If a is true, b has no effect on the value of the whole expression.

Expression Evaluation

- Short-circuiting
  - Other similar situations

    if (b != 0 && a/b == c) ...

    if (*p && p->foo) ...

    if (f || messy()) …
  - Can be avoided to allow for side effects in the condition functions
- Short-circuit evaluation
  - If the value of the expression does not depend on b, the evaluation of b is skipped.
  - This is useful, both in terms of optimization and semantically.

Expression Evaluation

- Expression-oriented vs. statement-oriented languages
  - expression-oriented (all statements are evaluated to a value):
    - functional languages (Lisp, Scheme, ML)
    - logic programming (everything is evaluated to a boolean value: **`true`**, **`false`** or **`undefined`**-Algol-68)
  - statement-oriented: some statements do not return anything
    - most imperative languages (e.g., **print** method returns **void**)
  - C is halfway in-between (some statements return values)
    - allows expressions to appear instead of statements and vice-versa:

```
if (a == b) {
    /* do the following if a equals b */

if (a = b) {
    /* assign b into a and then do
       the following if the result is nonzero */
```

    - C lacks a separate Boolean type: accepts an integer
      – if 0 then false, any other value is true.

Structured and Unstructured Flow

- Structured programming:
  - Makes extensive use of subroutines, block structures, for and while loops
  - Aimed at improving the clarity, quality, and development time of a computer program
- Unstructured programming:
  - Makes extensive use of subroutines, block structures, for and while loops
  - Makes use of GOTO statement, which could lead to "spaghetti code" that is difficult to follow and maintain

Goto Statements

- Control flow in assembly languages is achieved by means of conditional and unconditional jumps
- Unconditional jump: GOTO statements

```
10        PRINT "HELLO"
20        GOTO 10
```

- Conditional jump

JNZ	op1	jump if not zero
JE	op1 = op2	jump if equal
JNE	op1 != op2	jump if not equal
JG	op1 > op2	jump if greater than

- GOTO are hard to analyze
- Not limited to nested types
- Modern languages do not allow GOTO

Structured Alternatives to Goto

- Multilevel return
  - Return from a surrounding routine
  - permit a goto to branch to a lexically visible label outside the current subroutine

```
...
for ... (*iterate over lines*)
  ...
  if found(key, line) then begin
  rtn := line;
  goto 100;
  end;
...
100: return rtn;
```

- Exceptions
- call-with-current-continuation function
  - Capture a computation which is a state of the call stack
  - Resume same state at a later time

Continuation in Scheme

```
(define the-continuation #f)
 (define (test)
   (let ((i 0))
     ; call/cc calls its first function argument, passing a continuation variable the-continuation
     (call/cc (lambda (k) (set! the-continuation k)))
     ; The next time the-continuation is called, we start here.
     (set! i (+ i 1))
     i))
```

> (test)
1
> (the-continuation)
2
> (the-continuation)
3
; stores the current continuation (which will print 4 next) away
> (define another-continuation the-continuation)
> (test) ; resets the-continuation
1
> (the-continuation)
2
> (another-continuation) ; uses the previously stored continuation
4

Sequencing

- Sequencing
  - specifies a linear ordering on statements
    - one statement follows another
  - Some languages might waive this for optimization:
  - a = foo()
  - b = bar()
  - return a + b
  - The first two instructions can be executed sequentially, OR in reverse order OR even concurrently if foo and bar do not have side-effects.
  - E.g. Lisp

Sequencing

Issue: What's the value of a sequence of expressions/statements?

- The value of the last subexpression (most common)

  C: `a = 4, b = 5;` $\Longrightarrow 5$

  LISP: `(progn (setq a 4) (setq b 5))` $\Longrightarrow 5$

- The value of the first subexpression

  LISP: `(prog1 (setq a 4) (setq b 5))` $\Longrightarrow 4$

- The value of the second subexpression

  LISP: `(prog2 (setq a 4) (setq b 5) (setq c 6)` $\Longrightarrow 5$

Selection

- Selection statement types (in increasing convenience):
  - –If
  - –If/Else - no repeat negating condition.
  - –If/Elif/Else - don't require nesting (keep terminators from piling up at the end of nested if statements)
- Switch-Case statement.
  - –Can use array/hash table to look up where to go to,
  - –Can be more efficient than having to execute lots of conditions.
- Short-circuit evaluation of statements:

  if foo() or bar(): …
  - –We can short-circuit evaluation: if foo() is true, bar() is not called

Selection Code Generation

```
if ((A > B) and (C > D)) or (E <> F):
```

No short circuit

```
        r1 := A
        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          (JZ r1, L2)
L1: then clause  (label not actually used)
        goto L3
L2: else clause
L3:
```

Short circuit

```
        r1 := A
        r2 := B
        if r1 <= r2 goto L4  (JLE r1,r2,L4)
        r1 := C
        r2 := D
        if r1 > r2 goto L1    (JG r1,r2,L1)
L4: r1 := E
        r2 := F
        if r1 = r2 goto L2    (JE r1,r2,L2)
L1: then clause
        goto L3
L2: else clause
L3:
```

Selection

```
CASE ... (* potentially complicated expression *) OF
      1: clause A
      | 2, 7: clause B
      | 3..5: clause C
      | 10: clause D
      ELSE clause E
END
```

- - Less verbose,
- - More efficient than:
- `IF` (* potentially complicated expression *) `== 1 THEN`
- `    clause A`
- `ELSIF` (* potentially complicated expression *) `IN 2,7 THEN`
- `    clause B`
- `ELSIF` …

Iteration

- Enumeration-controlled loops:
  - Example: for-loop
  - One iteration per element in finite set
  - The number of iterations is known in advance.
- Logically controlled loops:
  - Example: while-loop
  - Executed until a Boolean condition changes
  - The number of iterations is not known in advance

Trade-offs in Iteration Constructs

- **Code Generation for** for**-Loops:**

```
        r1 := first
        r2 := step
        r3 := last
L1:  if r1 > r3 goto L2
        . . . – – loop body; use r1 for i
        r1 := r1 + r2
        goto L1
L2:
```

Is this efficient?

Trade-offs in Iteration Constructs

- **Code Generation for** for**-Loops:**

```
      r1 := first
      r2 := step
      r3 := last
      goto L2
L1:  . . . - - loop body; use r1 for i
      r1 := r1 + r2
L2:  if r1 ≤ r3 goto L1
```

  – Faster implementation because each of the iteration's contains a single conditional branch, rather than a conditional branch at the top and an unconditional jump at the bottom.

Iteration

- Iterator: pull values from the iterator object

```
for i in range(0, 101, 10):  # Python
      …
```

  – User can usefully define his own iterator object which makes it possible to iterate over other things:

```
for (Iterator<Integer> it =
          myTree.iterator(); it.hasNext();) {
    Integer i = it.next();
      System.out.println(i);
}
```

Iteration

- ***Post-test Loops:***
  - Test terminating condition at the bottom of the loop

```
repeat
        readln(line);
until line[1] = '$';
```

instead of

```
readln(line);
while line[1] <> '$' do
        readln(line);
```

- ***Midtest Loops:***

–Iteration often allows us to escape the block:

- `continue`
- `break`

```
for (;;) {
   readln(line);
   if (all_blanks(line)) break;
   consume_line(line);
}
```

Recursion

- Recursion:
  - –equally powerful to iteration
  - –mechanical transformations back and forth
  - –**often more intuitive** (sometimes less)
  - –naive implementation is less efficient than iteration:
    - **Stack frame allocations at every step**: copying values is slower than updates in iterations
  - –advantages of recursion:
    - fundamental to functional languages like Scheme
    - no special syntax required

Recursion

- Tail recursion
  - No computation follows recursive call

```
def factorial(n):
  if n == 0: return 1
  else: return factorial(n-1) * n

def tail_factorial(n, accumulator=1):
  if n == 0: return accumulator
  else: return tail_factorial(n-1, accumulator * n)
```

Applicative and Normal-Order Evaluation

- Applicative-order evaluation
  - Arguments are evaluated before a subroutine call
  - Default in most programming languages
- Normal-order evaluation
  - Arguments are passed to the subroutine unevaluated.
  - The subroutine evaluates them as needed.
  - Useful for infinite or lazy data structures that are computed as needed.
  - Example: macros in C/C++

Lazy Evaluation

- Lazy evaluation
- Evaluate expressions when their values are needed.
- Cache results to avoid recomputation.
- Python

```
>>> r = range(10)
>>> print(r)
range(0, 10)
>>> print(r[3])
3
```