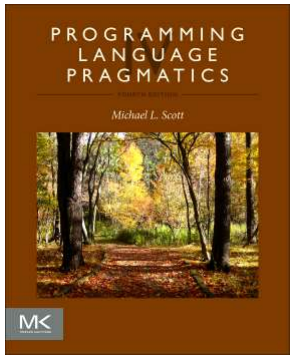
Chapter 6:: Control Flow

- 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
```

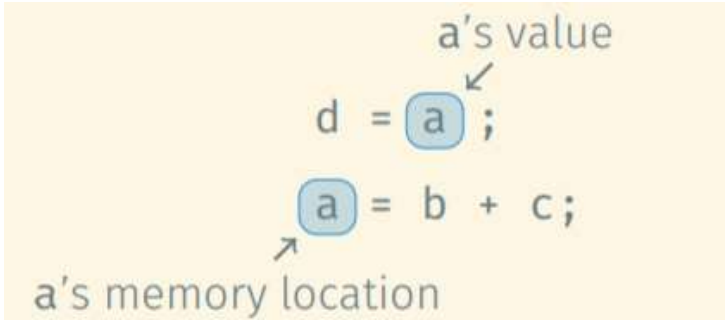
Definite Assignment

- 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

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

Value model vs. reference model

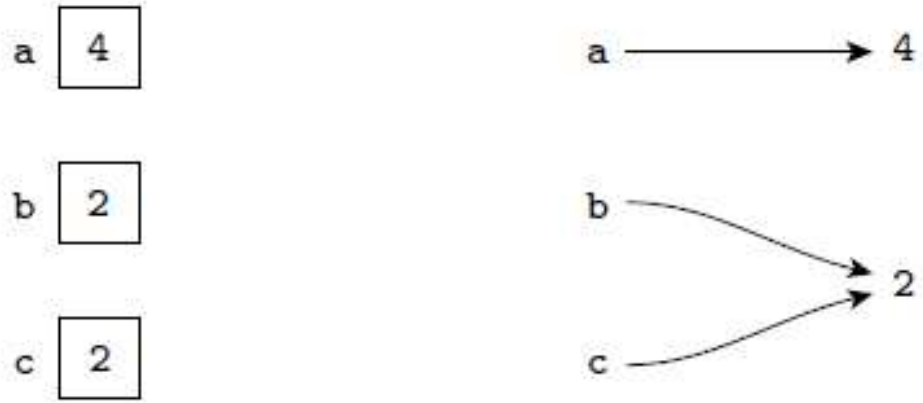


Figure 6.2 The value (left) and reference (right) models of variables. Under the reference model, it becomes important to distinguish between variables that refer to the same object and variables that refer to different objects whose values happen (at the moment) to be equal.

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.

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);
}
```