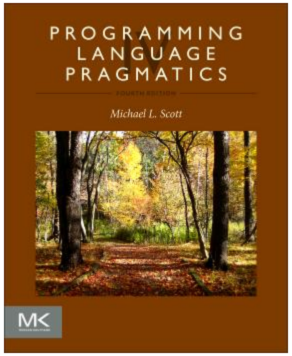
Chapter 6:: Control Flow

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Copyright © 2016 Elsevier

Introduction

- Core issues in language design
- The issue of *control flow* or *ordering in program execution*
  - – determines what should be done first, what second, and so forth, to accomplish some desired task.
- Order of execution matters for statements, and for expressions with *side effects*.

Expression Evaluation

- An expression is any legal combination of symbols that represents a value.
- An expression generally consists of either
  - a simple object (e.g., a literal constant, or a named variable or constant) or
  - an operator or function applied to a collection of operands or arguments, each of which in turn is an expression.
- Example: x + 5
  - + operator (built-up function)
  - x and 5: operands (arguments of an operator)

Expression Evaluation

- A language may specify that function calls employ prefix, infix, or postfix notation.

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

- Most imperative languages use infix notation for binary operators and prefix notation for unary operators and (with parentheses around the arguments) other functions.

ELSEVIER

Expression Evaluation

- **Precedence rules** specify that certain operators, in the absence of parentheses, group “more tightly” than other operators.
- Levels of precedence for several well-known languages (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
  - **Lesson**: When unsure, use parentheses!

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

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

Expression Evaluation

```
#include <stdio.h>
int main()
{       int a = 4;
        int b = 1;
        int c = ++a-b;
        printf("c: %d \n", c);
        printf("a: %d  \n", a);
        printf("b: %d  \n", b);
        return 0;
}
```

```
c:
a:
b:
```

Expression Evaluation

```
#include <stdio.h>
int main()
{       int a = 4;
        int b = 1;
        int c = ++a-b;
        printf("c: %d \n", c);
        printf("a: %d  \n", a);
        printf("b: %d  \n", b);
        return 0;
}
```

```
c: 4
a: 5
b: 1
```

Expression Evaluation

```
a = 4
b = 1
a += 1
c = a - b
print("c:", c)
print("a:", a)
print("b:", b)
```

```
c:
a:
b:
```

Expression Evaluation

```
a = 4
b = 1
a += 1
c = a - b
print("c:", c)
print("a:", a)
print("b:", b)
```

```
c: 4
a: 5
b: 1
```

Expression Evaluation

```
#include <stdio.h>
int main()
{      int a = 4;
       int b = 1;
       int c = a-b++;
       printf("c: %d \n", c);
       printf("a: %d  \n", a);
       printf("b: %d  \n", b);
       return 0;
}
```

c:

a:

b:

Expression Evaluation

```c
#include <stdio.h>
int main()
{       int a = 4;
        int b = 1;
        int c = a-b++;
        printf("c: %d \n", c);
        printf("a: %d  \n", a);
        printf("b: %d  \n", b);
        return 0;
}
```

```
c: 3
a: 4
b: 2
```

Expression Evaluation

```
#include <stdio.h>
int main()
{       int a = 4;
        int b = 1;
        int c = a*++b;
        printf("c: %d \n", c);
        printf("a: %d  \n", a);
        printf("b: %d  \n", b);
        return 0;
}
```

```
c:
a:
b:
```

Expression Evaluation

```
#include <stdio.h>
int main()
{     int a = 4;
      int b = 1;
      int c = a*++b;
      printf("c: %d \n", c);
      printf("a: %d  \n", a);
      printf("b: %d  \n", b);
      return 0;
}
```

```
c: 8
a: 4
b: 2
```

ELSEVIER

Expression Evaluation

```
#include <stdio.h>
int main()
{       int a = 4;
        int b = 1;
        int c = a++*b-1;
        printf("c: %d \n", c);
        printf("a: %d  \n", a);
        printf("b: %d  \n", b);
        return 0;
}
```

c:

a:

b:

Expression Evaluation

```
#include <stdio.h>
int main()
{      int a = 4;
       int b = 1;
       int c = a++*b-1;
       printf("c: %d \n", c);
       printf("a: %d  \n", a);
       printf("b: %d  \n", b);
       return 0;
}
```

```
c: 3
a: 5
b: 1
```

Expression Evaluation

- **Associativity rules** specify whether sequences of operators equal precedence group to the right or left.
- The basic arithmetic operators almost always associate left-to-right, so 9 - 3 - 2 is 4 and not 8.
  - In Fortran, the exponentiation operator (**) associates right-to-left, so 4**3**2 is 262144 and not 4096.
  - In Ada, exponentiation does not associate, one must write either (4**3)**2 or 4**(3**2).

Expression Evaluation

- **Associativity rules (cont.)**
  - **Lesson**: Rules for precedence and associativity vary so much from one language to another. When unsure, use parentheses!
    - Precedence and associativity rules can be overridden with parentheses.

ELSEVIER

Expression Evaluation

- **Expression-oriented languages:**
  - Expressions are the building blocks of programs
  - Computation consists entirely of expression evaluation
  - Functional languages (Lisp, Scheme, ML)
- **Statement-oriented languages:**
  - Computation typically consists of an ordered series of changes to the values of variables in memory
  - Assignments provide the principal means by which to make the changes
  - Most imperative languages

ELSEVIER

Expression Evaluation

- **Assignment**
  - Statement (or expression) executed for its side effect
  - Executed solely for their *side effects*

Expression Evaluation

- **Side Effects**
  - A programming language construct is said to have a side effect if it influences subsequent computation (and ultimately program output) in any way other than by returning a value for use in the surrounding context.
  - Assignment is perhaps the most fundamental side effect:
    - They change the value of a variable
      - Thereby influencing the result of any later computation in which the variable appears

Expression Evaluation

- **Side Effects**
  - Often discussed in the context of functions
  - When a function changes a two-way parameter or a non-local variable
  - Side effects are fundamental to the whole Von Neumann model of computing

```cpp
#include <iostream>

using namespace std;

int total=0;

// function sum changes non-local variable
int sum(int x, int y)
{
    total = x + y;
    return total;

}

int main()
{

    sum (100, 200);
    cout << "Total: " << total << endl;

    return 0;
}
```

* Sebesta

Expression Evaluation

- **Side Effects**
  - Often discussed in the context of functions
  - When a function changes a two-way parameter or a non-local variable
  - Side effects are fundamental to the whole Von Neumann model of computing

```
x = 0
def foo():
    global x
    x += 5
    return x

a = foo() + x + foo()
print(a)
print(x)
```

* Sebesta

Expression Evaluation

- **Side Effects**
  - Often discussed in the context of functions
  - When a function changes a two-way parameter or a non-local variable
  - Side effects are fundamental to the whole Von Neumann model of computing

```
int x = 0;
int foo() {
  x+=5;
  return x;
}

int a = foo()+ x + foo();
cout << a;
cout << x;
```

* Sebesta

ELSEVIER

Expression Evaluation

- Many imperative languages distinguish between expressions and statements
  - Expressions always produce a value, and may or may not have side effects
  - Statements are executed solely for their side effects, and return no useful value
- Imperative programming is sometimes described as "computing by means of side effects."

ELSEVIER

Expression Evaluation

- Purely functional languages have no side effects
  - Haskell and Miranda are purely functional
  - Many other languages are mixed:
    - ML and Lisp are mostly functional, but make assignment available to programmers who want it
    - C#, Python, and Ruby are mostly imperative, but provide a variety of features (e.g., first-class functions, polymorphism, functional values) that allow them to be used in a largely functional style

ELSEVIER

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
  - A compiler that performs *short-circuit evaluation* of Boolean expressions will generate code that skips the second half of both of these computations when the overall value can be determined from the first half.

```c
// Short-circuiting

#include <stdio.h>

int main()
{
    int a = 5;
    int b = -10;

    // Here b == -10 is not evaluated because a!= 5 is false

    if (a != 5 && b == -10) {
        printf("This won't be printed!\n");
    }
    else {
        printf("Else block is printed.");
    }

    return 0;
}
```

```
Else block is printed.

...Program finished with exit code 0
Press ENTER to exit console.
```

Expression Evaluation

- **Short-circuiting**
  - Short-circuit evaluation can save significant amounts of time in certain situations.
    - if (very_unlikely_condition && very_expensive_function()) ...
  - There are situations in which short circuiting may not be appropriate.
  - If expressions E1 and E2 both have side effects, we may want the conjunction E1 and E2 (and likewise E1 or E2) to evaluate both halves.

ELSEVIER

Sequencing

– Specifies a linear ordering on statements
  - One statement follows another
– List of statements can be enclosed with
  – begin …. end
  – { …. }
– Such a delimited list is usually called a *compound statement* - *Block*

Selection

- A selection statement provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Selection

- Two-way selectors

  if control_expression

  then clause

  else clause

Selection

- Multiple-way selectors
  - C, C++, Java, and JavaScript

```
switch (expression) {
   case const_expr1: stmt1;
   …
   case const_exprn: stmtn;
   [default: stmtn+1]
}
```

Selection

- Multiple-way selectors
  - Can an appear as direct extensions to two-way selectors, using else-if clauses, for example in Python:

```
if count < 10 :
    bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

```cpp
#include <iostream>

using namespace std;

int main()
{

   int c;
   cout << "What grade are you in? "<< endl;
   cin >> c;

   switch(c) {
      case 1:
         cout << "Freshman! ";
         break;
      case 2:
         cout << "Sophomore! ";
         break;
      case 3:
         cout << "Junior! ";
         break;
      case 4:
         cout << "Senior! " ;
         break;
      default:
         cout << "Hmmm... Experienced? :) ";
         break;
   }
   return 0;
}
```

```
What grade are you in?
2
Sophomore!
```

Iteration & Recursion

- *Iteration* and *recursion* allow to perform similar operations repeatedly.
- Iteration
  - Programmers in imperative languages tend to use iteration
  - More "natural" in imperative languages, because it is based on the repeated modification of variables
- Recursion
  - More common in functional languages
  - More "natural" in functional languages, because it does *not* change variables

ELSEVIER

Iteration

- In most languages, iteration takes the form of *loops*.
- Two principal varieties of loops:
  - –Enumeration-Controlled Loops
    - "For every element of set"
  - –Logically Controlled Loops
    - "While condition is true"

ELSEVIER

Iteration

- Enumeration-Controlled Loops
  - Executed once for every value in a given finite set
  - The number of iterations is known before the first iteration begins
  - `for loop` (general term)
  - ```
    for (i=0; i<5; i++){
     // … }
    ```
  - ```
    for x in range(2, 6):
      print(x)
    ```

ELSEVIER

Iteration

- Logically Controlled Loops
  - Executed until some Boolean condition (which must generally depend on values altered in the loop) changes value
  - `while (`*`condition`*`)`
    `do `*`statement`* (general)

Iteration

- ```
  int i = 1;
  while (i < 6){
      cout<<i<<endl;
      i++;
  }
  ```

- ```
  int i = 0;
  do {
      System.out.println(i);
      i++;
  }
   while (i < 10);
  ```

Recursion

- Equally powerful to iteration
- Mechanical transformations back and forth
  - Functions can call themselves, or call other functions that then call them back in turn
- Is iteration more efficient than recursion?
  - "*Naïve*" implementation of iteration is usually more efficient than "*naïve*" implementation of recursion.
- As efficient in cases where you can use *tail recursion*

ELSEVIER

Recursion

- Tail recursion
  - A tail-recursive function ends by the returning value of the recursive call
  - 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);
}
```

Recursion

- call f (3)
- call 3 * f (2)
- call 2 * f (1)
- call 1 * f (0)
- return 1
- return 1
- return 2
- return 6

```cpp
#include <iostream>

using namespace std;

int f (int n)
{
    if (n <= 0){

    return 1;

    }

    return n*f(n-1);
}

int main()
{
    cout << f(3) << endl;
    return 0;
}
```

Recursion

- replace arguments with (3, 1)
- replace arguments with (2, 3)
- replace arguments with (1, 6)
- replace arguments with (0, 6)
- return 6

```cpp
#include <iostream>

using namespace std;

int f_tail_recursion (int n, int i)
{
    if (n <= 0){

    return i;

    }

    return f_tail_recursion(n-1, i*n);
}

int main()
{
    cout << f_tail_recursion(3, 1) << endl;
    return 0;
}
```

References

- M. L. Scott, Programming Language Pragmatics (4th Edition), Morgan Kaufmann, 2016.
- R. W. Sebesta, Concepts of Programming Languages (11th Edition), Pearson, 2016.