# Chapter 6

Control Flow: Expressions

## Expressions

Implementation involves fetching operands and executing operations

Precedence and associativity of operations are primary design issues

C and C++ have over 50 operators and 15 levels of precedence

Recall syntax directed semantics in BNF rules for arithmetic expressions

| 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 operators at the top of the figure group most tightly.



## Operand Evaluation & Side Effects

Consider the following C++ example:

```
int a[20];  // 20 element array
int i = 10;
a[i] = i++;
```

What value gets stored where???

What if the last statement is changed to:

```
a[i] = ++i;
```

or

```
a[i++] = i;
```

## Operand Evaluation & Side Effects

Another C++ example

```
int x = 5;
int y = x + f(x);
```

```
// x uses pass by reference
int f(int & x) {
   int t = x;
   x = x - 5;
   return 2 * t;
}
```

## Operand Evaluation & Side Effects

Some optimizing compilers rearrange/reorder operand evaluation to generate more efficient object code

Ada does not allow functions to create side effects (no OUT parameters allowed)

## Boolean Expressions

Precedence levels of Boolean operators (and relational operators) varies among languages

```
A or B and C
```

- Usually "and" higher than "or"
- Ada doesn't do this
- Usually higher than relational operators (But not in Pascal: `A < 5 or B`)

## Boolean Expressions

C (and early versions of C++) is the only popular imperative language without a boolean type

- 0 means false
- Non-zero means true

So the following are legal Boolean expressions:

```
8 < 6 < 4     (true)
5 == 5 == 5   (false)
```

And what about Ruby?

## Short Circuiting

Expression evaluation in which the result can be determined without evaluating all operands and operators

Examples

```
if ( a != 0 && b/a < 10) ➔ good
if ( a != 0 || b++ > 5)  ➔ bad
```

## Boolean Expressions

Pascal does not have short circuit Boolean expressions

Ada provides both forms

**and/or** for non-short circuit

**and then/or else** for short circuit

C/Java have short circuit Boolean expressions only

&& (and), || (or)

They also have bitwise logical operations that do not short circuit

& (bitwise and), | (bitwise or), ^ (bitwise xor)

## Ternary Operators

C, C++, Java conditional expression

```
x = a + ((b < c) ? b : c);
```

Perl/Ruby: 3 – way comparison <=>

| | |
|---|---|
| -1 if 1st < 2nd | `5 <=> 10` |
| 0 if 1st = 2nd | `5 <=> 5` |
| +1 if 1st > 2nd | `5 <=> 1` |

## Assignment Expressions

In the C/Java languages assignment is an expression

Makes the following possible:

```
int x, y, z;
x = y = z = 4;
z = 5 + (x = y-1);
if ((x = y) == 4) { … }
while ((ch = getchar())!=EOF) { … }
```