Chapter 6

Arithmetic Expressions

- Their evaluation was one of the motivations for the development of the first programming languages

- Arithmetic expressions consist of operators, operands, parentheses, and function calls

Design issues for arithmetic expressions:

1. What are the operator precedence rules?
2. What are the operator associativity rules?
3. What is the order of operand evaluation?
4. Are there restrictions on operand evaluation side effects?
5. Does the language allow user-defined operator overloading?
6. What mode mixing is allowed in expressions?

Chapter 6

A *unary operator* has one operand

A *binary operator* has two operands

A *ternary operator* has three operands

Def: The *operator precedence rules* for expression evaluation define the order in which adjacent operators of different precedence levels are evaluated
(adjacent means they are separated by at most one operand)

- *Typical precedence levels*

1. **parentheses**
2. **unary operators**
3. **** (if the language supports it)**
4. ***, /**
5. **+, -**

Def: The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated

Chapter 6

- Typical associativity rules:
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., FORTRAN)

- APL is different; all operators have equal precedence and all operators associate right to left

Precedence and associativity rules can be overriden with parentheses

Operand evaluation order

- *The process:*
  1. *Variables:* just fetch the value
  2. *Constants:* sometimes a fetch from memory; sometimes the constant is in the machine language instruction
  3. *Parenthesized expressions:* evaluate all operands and operators first
  4. *Function references:* The case of most interest!
     - Order of evaluation is crucial

Functional side effects - when a function changes a two-way parameter or a nonlocal variable

Chapter 6

The problem with functional side effects:

- When a function referenced in an expression alters another operand of the expression e.g., for a parameter change:

```
        a = 10;
        b = a + fun(&a);
/* Assume that fun changes its parameter */
```

Two Possible Solutions to the Problem:

1. Write the language definition to disallow functional side effects

- No two-way parameters in functions
- No nonlocal references in functions

- *Advantage:* it works!
- *Disadvantage:* Programmers want the flexibility of two-way parameters (what about C?) and nonlocal references

2. Write the language definition to demand that operand evaluation order be fixed

- *Disadvantage*: limits some compiler optimizations

Chapter 6

Conditional Expressions

- C, C++, and Java (`?:`)

  e.g.

```
average = (count == 0)? 0 : sum / count;
```

Operator Overloading

- Some is common (e.g., `+` for `int` and `float`)

- Some is potential trouble (e.g., `*` in C and C++)

  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Can be avoided by introduction of new symbols (e.g., Pascal s `div`)

- C++ and Ada allow user-defined overloaded operators

  Potential problems:

  - Users can define nonsense operations
  - Readability may suffer

Chapter 6

Implicit Type Conversions

Def: A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type

Def: A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type

Def: A *mixed-mode* expression is one that has operands of different types

Def: A *coercion* is an implicit type conversion

- The disadvantage of coercions:

- They decrease in the type error detection ability of the compiler

- In most languages, all numeric types are coerced in expressions, using widening conversions

- In Modula-2 and Ada, there are virtually no coercions in expressions

Chapter 6

Explicit Type Conversions

- Often called *casts*

e.g.
Ada:

```
FLOAT(INDEX)  -- INDEX is INTEGER type
```

C:

```
(int)speed    /* speed is float type */
```

Errors in Expressions

- Caused by:

  - Inherent limitations of arithmetic
    e.g. division by zero

  - Limitations of computer arithmetic
    e.g. overflow

- Such errors are often ignored by the run-time system

Chapter 6

Relational Expressions

- Use relational operators and operands of various types
- Evaluate to some boolean representation
- Operator symbols used vary somewhat among languages (`!=`, `/=`, `.NE.`, `<>`, `#`)

Boolean Expressions

- Operands are boolean and the result is boolean

- *Operators:*

FORTRAN 77	*FORTRAN 90*	*C*	*Ada*
`.AND.`	`and`	`&&`	`and`
`.OR.`	`or`	`||`	`or`
`.NOT.`	`not`	`!`	`not`
			`xor`

- C has no boolean type--it uses `int` type with 0 for false and nonzero for true

- One odd characteristic of C s expressions: `a < b < c` is a legal expression, but the result is not what you might expect

Chapter 6

Precedence of All Operators:

Pascal: `not`, **unary** `-`
`*`, `/`, `div`, `mod`, `and`
`+`, `-`, `or`
relops

Ada: **`**`**
`*`, `/`, `mod`, `rem`
unary `-`, `not`
`+`, `-`, `&`
relops
`and`, `or`, `xor`

C, C++, and Java have over 50 operators and 17 different levels of precedence

Short Circuit Evaluation

Pascal: **does not use short-circuit evaluation**
Problem: table look-up

```
index := 1;
  while (index <= length) and
         (LIST[index] <> value) do
    index := index + 1
```

Chapter 6

C, C++, and Java: **use short-circuit evaluation for the usual Boolean operators (`&&` and `||`), but also provide bitwise Boolean operators that are not short circuit (`&` and `|`)**

Ada: **programmer can specify either (short-circuit is specified with `and then` and `or else`)**

FORTRAN 77: **short circuit, but any side-affected place must be set to undefined**

Short-circuit evaluation exposes the potential problem of side effects in expressions
e.g. `(a > b) || (b++ / 3)`

Assignment Statements

The operator symbol:

1. `=` **FORTRAN, BASIC, PL/I, C, C++, Java**
2. `:=` **ALGOLs, Pascal, Modula-2, Ada**

`=` **can be bad if it is overloaded for the relational operator for equality**

e.g. (PL/I) `A = B = C;`

Note difference from C

Chapter 6

More complicated assignments:

1. Multiple targets **(PL/I)**

```
A, B = 10
```

2. Conditional targets **(C, C++, and Java)**

```
(first = true) ? total : subtotal = 0
```

3. Compound assignment operators **(C, C++, and Java)**

```
sum += next;
```

4. Unary assignment operators **(C, C++, and Java)**

```
a++;
```

C, C++, and Java treat = as an arithmetic binary operator

e.g.

```
a = b * (c = d * 2 + 1) + 1
```

This is inherited from ALGOL 68

Chapter 6

Assignment as an Expression

- In C, C++, and Java, the assignment statement produces a result
  - So, they can be used as operands in expressions

e.g. `while ((ch = getchar() != EOF) { ... }`

Disadvantage
- Another kind of expression side effect

Mixed-Mode Assignment

- In FORTRAN, C, and C++, any numeric value can be assigned to any numeric scalar variable; whatever conversion is necessary is done

- In Pascal, integers can be assigned to reals, but reals cannot be assigned to integers (the programmer must specify whether the conversion from real to integer is truncated or rounded)

- In Java, only widening assignment coercions are done

- In Ada, there is no assignment coercion