Chapter 6 - Expressions and Assignment Statements

- Their evaluation was one of the motivations for the development of the first PLs
- 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?

Operator Precedence

- A *unary operator* has one operand
- A *binary operator* has two operands
- A *ternary operator* has three operands

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. +, -

Operator Associativity

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

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

Use of Parantheses

Precedence and associativity rules can be overridden 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.

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

Conditional Expressions

C, C++, and Java

- ?: ternary operator

```
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

Implicit Type Conversions

- A ***narrowing conversion*** is one that converts an object to a type that cannot include all of the values of the original type
- 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
- A ***mixed-mode* expression** is one that has operands of different types.
- 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

Explicit Type Conversions

Often called ***casts***

- e.g. Ada:
  `FLOAT(INDEX) -- INDEX is an INTEGER`
- e.g. 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

Relational Expressions

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

Relational operators always have lower precedence than the arithmetic operators:
e.g., `a+1 > 2*b` = `(a+1) > (2*b)`

Boolean Expressions

- Operands are Boolean and the result is Boolean.
- *Operators:*

F77	***F90***	***C***	***Ada***
`.AND.`	`and`	`&&`	`and`
`.OR.`	`or`	`||`	`or`
`.NOT.`	`not`	`!`	`not`
			`xor`

Common Precedence order: NOT, AND, OR.

Ada: logical ops have same precedence and no assoc!

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

no Boolean ⇒ low readablity, lost error detection

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

Precedence of All Operators

Arith.exprs can be operands of rel. exprs and rel.exprs can be operands of Boolean exprs.

Common Precedence order:
Arithmetic, relative, logical

Pascal: *(boolean exprs. has higher precedence than rel exprs.)*

```
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

Result is determined without evaluating all of the operands and/or operators.
*e.g. (13 * A) * (B / 13 - 1) or (A >= 0) and (B < 10)*

Pascal: does not use short-circuit evaluation
⇒ Problem: table look-up using `while` statement.

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

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`**)
***This is the best design choice!*

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

- provides a mechanism by which the user can dynamically change the bindings of values to variables.

The operator symbol:

= FORTRAN, BASIC, PL/I, C, C++, Java
⇒ can be bad if = is overloaded for the relational operator for equality
e.g. (PL/I) `A = B = C;`

:= ALGOLs, Pascal, Modula-2, Ada

The assignment operator in C and C++ is treated much like a binary operator, and as much it can appear embedded in expressions.

We will see the other design choices...

How Assignments are Used?

More complicated assignments:

1. Multiple targets
`A, B = 10` (PL/I) (can be simulated in C)

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) (prefix or postfix)
`a++;`

C/C++/Java treat = as an arithmetic binary operator:
e.g. `a = b * (c = d * 2 + 1) + 1`

This is inherited from ALGOL 68

When two unary operators apply to the same operand, the association is right to left:
e.g. `- count ++`

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
  - Difficult to read and understand
    e.g., `a = b + (c = d / b++) - 1`

C's assignment operator allows the effect of multiple-target assignments:
e.g., `sum = count = 0;`

Low safety of = operator in C/C++:

Loss of error detectin in the C design of the assignment operation that frequently leades to program errors:
e.g., `if (x = y) ...`

Mixed-Mode Assignment

- In FORTRAN, C, and C++, any numeric value can be assigned to any numeric scalar ariable; whatever conversion is necessary is done
- In Pascal, integers can be assigned to reals, but not vice versa (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.