Arithmetic Expressions

In Text: Chapter 7

Outline

- Precedence
- Associativity
- Evaluation order/side effects
- Conditional expressions
- Type conversions and coercions
- Assignment
### Arithmetic Expressions

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

- Design issues for arithmetic expressions:
  - What are the operator precedence rules?
  - What are the operator associativity rules?
  - What is the order of operand evaluation?
  - Are there restrictions on operand evaluation side effects?
  - Does the language allow user-defined operator overloading?
  - What mode mixing is allowed in expressions?

### Operators

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

- Functions can be viewed as unary operators with an operand of a simple list.
- Argument lists (and parameter lists) treat separators (comma, space) as "stacking" or "append" operators.
- Keyword in a language statement as functions in which the remainder of the statement is the operand.

- **Operator precedence** and operator **associativity** are important considerations.
### Operator Precedence

The **operator precedence rules** for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated:

- Parenthetical groups (...)
- Exponentiation **
- Mult & Div *, /
- Add & Sub + , -
- Assignment :=

\[ A * B + C ** D / E - F \] \(\Rightarrow\) \( ( (A * B) + ((C ** D) / E) - F) \)

**QUESTIONS:**
1. Where do functions come in the hierarchy?
2. Where do unary operators lie?
3. Where do logical and boolean operators lie?

---

### Operator Associativity

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

- Left associative *, /, +, -
- Right associative **

\[ B ** C ** D - E + F * G / H \] \(\Rightarrow\) \( (((B ** (C ** D)) - E) + ((F * G) / H)) \)

**EFFECTIVELY**
- Most programming languages evaluate expressions from left to right
- LISP uses parentheses to enforce evaluation order
- APL is strictly RIGHT to LEFT, taking note only of parenthetical groups
Chapter 7: Arithmetic Expressions

Operand Evaluation Order

- Order of evaluation is crucial

A = B + C
- Get value for B, get value for C, add the values
- Get value for C, Get value for B, add the values

- Function references is when order of evaluation is most crucial
  - Functional side-effects

Side Effects

- Functional side effects — when a function changes a two-way parameter (pass-by-reference) or a non-local variable

- The problem with functional side effects:
  - When a function referenced in an expression alters another operand of the expression

- Example, for a parameter passed by reference:
  
  ```
  a = 10;
  b = a + fun(&a);
  /* Assume that fun changes its param */
  ```
Side Effects
(Non-Local Reference)

Procedure sub 1 (...);
    var a : integer;
    function fun (x : integer) : integer;
        x := a + 2;
        return (x)
    end;
    a := 7;
    b := a + fun ( a );
    print ( a , b );
end;

w/o SE:  7 , 16
w/ SE:  9 , 16    a , fun(a)
w/ SE:  9 , 18    fun(a) , a

Solutions for Side Effects

Two Possible Solutions to the Problem

1. Write the language definition to disallow functional side effects
   - No two-way parameters in functions
   - No non-local references in functions
   - Advantage: it works!
   - Disadvantage: Programmers want the flexibility

2. Write the language definition to demand that operand evaluation order be fixed
   - Disadvantage: limits some compiler optimizations
Operator Overloading

An operator or function is overloaded if its meaning depends on the number or types of its arguments

- $+$ (real, integer)
- $-$ (unary, binary)

Also called ad-hoc polymorphism

- Some is common (e.g., $+$ for int and float)
- Some is potential trouble (e.g., $\&$ in C)
  - $X = A \& B$ bitwise logical AND
  - $X = \& B$ address

Loss of compiler error detection
- omission of an operand should be a detectable error

C++ and Ada allow user-defined overloaded operators

Potential problems:
- Users can define nonsense operations
- Readability may suffer
Implicit Type Conversions

- A coercion is an implicit type conversion

```pascal
var x: integer;
    y, z: real;
...
y := x + z;    /* x is automatically converted to "real" */
```

"mixed-mode" expression

Implicit Type Conversions

- Implicit type conversion can be achieved by narrowing or widening one or more operands

- 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
Widening vs Narrowing

- complex
- double
- floating (E-notation)
- real (fixed point)
- integer
- boolean (represented by \{0,1\})

A widening conversion sequence is one in which NO information is lost (the preferred sequence)
A narrowing conversion sequence is one in which information is lost at each step downward

Question: Should narrowing involve rounding or truncation?

---

Widening vs Narrowing

- Given a choice it is better to widen if possible.
- Suppose: \(x\) and \(y\) are integers; \(z\) is a float
- want to evaluate \(x = y \times z\)

(a) the order of evaluation is: \((y \times z)\) then \((x := \text{result})\)

(b) So widen \((y \times z)\) to \(((\text{real } y) \times z)\) so as to force multiplication between two reals, yeilding a real result

(c) Narrow the assignment to \((x := \text{int result})\)

Example: \(y := 3; \ z := 5.9\)

\(x \leq 15\) if \(y\) is narrowed; \(x \leq 17\) if \(y\) is widen
Comments on Implicit Coercions

- They decrease 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

**Coercion is different from explicit type conversion**

Explicit Type Conversions

- Often called casts

- Ada example:
  ```
  FLOAT(INDEX)  -- INDEX is INTEGER type
  ```

- C example:
  ```
  (int) speed    /* speed is float type */
  ```
Relational Operators and Boolean Expressions

- Relational operators compare two operands and return a boolean
  - =   <   >   <=   >=   <>
- Lower precedence than arithmetic operators
  - \( a + b < c + d \equiv (a + b) < (c + d) \)
- Boolean values: true, false
- Boolean operators: and, or, xor, not, =
- True boolean values are helpful.
  - In C, use integers:
    - 0 = false; other = false
    - implications: \( a > b > c \) is legal!

Precedence of All Operators

- Pascal: not, unary -
  \( *, /, \text{div}, \text{mod}, \text{and} \)
  \( +, -, \text{or} \)
  relops
- Ada:
  \( ** \)
  \( *, /, \text{mod}, \text{rem} \)
  unary -, not
  \( +, -, \& \)
  relops
  and, or, xor
- C, C++, and Java have > 50 operators and 17 different precedence levels
Short Circuit Evaluation

- Stop evaluating operands of logical operators once result is known
- Get a result without evaluating entire expression.
  \[(x \text{ and } y) \equiv \text{if } x \text{ then } y \text{ else false}\]
  \[(x \text{ or } y) \equiv \text{if } x \text{ then true else y}\]
  \[(x \text{ and } y \text{ are arbitrary boolean expressions)}\]

Problem 1:
- Need to KNOW how the language works!
  ```
  index := 1;
  while (index <= length) and
    (LIST[index] <> value) do
    index := index + 1
  ```

Problem 2:
- Short-circuit evaluation exposes the potential problem of side effects in expressions
  C Example: \[(a > b) || (b++ / 3)\]
Short Circuit Evaluation

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

- Ada: programmer can specify either
  - ( x or else y )
  - ( x and then y )

Assignment Statements

- The operator symbol:
  - = FORTRAN, BASIC, PL/I, C, C++, Java
  - := ALGOLs, Pascal, Modula-2, Ada

  = can be bad if it is overloaded with the relational operator for equality
    (e.g. in PL/I, A = B = C; )
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++;

5. C, C++, and Java treat = as an arithmetic binary operator
   - a = b * (c = d * 2 + 1) + 1
   - This is inherited from ALGOL 68

Assignment as an Expression

- In C, C++, and Java, the assignment statement produces a result
- So, they can be used as operands in expressions
  - while ((ch = getchar() != EOF) { ... }
  - (b * (c = d * 2 + 1) + 1)

- 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.