7. Expressions and Assignment Statements

7.2 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
  - 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?
7.2 Arithmetic Expressions (continued)

- 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
  - parentheses
  - unary operators
  - ** (if the language supports it)
  - *, /
  - +, -

7.2 Arithmetic Expressions (continued)

- 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
- Precedence and associativity rules can be overridden with parentheses
- Operand evaluation order
  - The process:
    - Variables: just fetch the value
    - Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
    - Parenthesized expressions: evaluate all operands and operators first
    - Function references: The case of most interest!
    - Order of evaluation is crucial
7.2 Arithmetic Expressions (continued)

- 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 */
    ```
  - Same problem with global variables

7.2 Arithmetic Expressions (continued)

- Two Possible Solutions to the Problem:
  - 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
  - Write the language definition to demand that operand evaluation order be fixed
    - Disadvantage: limits some compiler optimizations

- Conditional Expressions
  - C, C++, and Java (?:) e.g.
    ```
    average = (count == 0)? 0 : sum / count;
    ```
### 7.3 Overloaded Operators

- 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)
  - Some loss of readability
  - 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, even when the operators make sense

### 7.4 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 e.g., float to int
- 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 e.g., int to float
- 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 Ada, there are virtually no coercions in expressions
7.4 Type Conversions (continued)

- **Explicit Type Conversions**
  - Often called casts e.g.
  - Ada:
    
    ```
    FLOAT(INDEX)  -- INDEX is INTEGER type
    ```
  - Java:
    
    ```
    (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

7.5 Relational and Boolean Expressions

- **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
7.5 Relational and Boolean Expressions (continued)

- **Precedence of all Ada Operators:**
  - `**`, abs, not
  - `*`, `/`, mod, rem
  - unary `-`, `+`
  - binary `+`, `-`, `&`
  - relops, in, not in
  - and, or, xor, and then, or else

- C, C++, and Java have over 40 operators and least 15 different levels of precedence

7.6 Short Circuit Evaluation

- Suppose Java did not use short-circuit evaluation
  - Problem: table look-up
    ```java
    index = 1;
    while (index <= length) && (LIST[index] != value)
        index++;
    ```

- 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)`
7.7 Assignment Statements

- **The operator symbol:**
  - `=` FORTRAN, BASIC, PL/I, C, C++, Java
  - `:=` ALGOLs, Pascal, 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

7.7 Assignment Statements (continued)

- More complicated assignments:
  - Multiple targets (PL/I)
    - `A, B = 10`
  - Conditional targets (C, C++, and Java)
    - `(first == true) ? total : subtotal = 0`
  - Compound assignment operators (C, C++, and Java)
    - `sum += next;`
  - 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
7.7 Assignment Statements (continued)

- **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

7.8 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