Expressions and Assignment Statements

Lecture 12

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Arithmetic Expressions

- Automatic evaluation of arithmetic expressions similar to those found in mathematics, science, and engineering was one of the primary goals of the first high-level programming languages.

- Most of the characteristics of arithmetic expressions in programming languages were inherited from conventions that had evolved in mathematics.

- In programming languages, arithmetic expressions consist of ***operators***, ***operands***, ***parentheses***, and ***function calls***.
  - An operator can be ***unary***, meaning it has a single operand
  - An operator can be ***binary***, meaning it has two operands
  - An operator can be ***ternary***, meaning it has three operands

Arithmetic Expressions

- ***Infix***, ***Postfix*** and ***Prefix*** notations are three different but equivalent ways of writing expressions.
  - Infix notation: X + Y
    - Operators are written in-between their operands.
    - This is the usual way we write expressions.
    - Example: A * (B + C) / D
  - Postfix notation (also known as "Reverse Polish notation"): X Y +
    - Operators are written *after* their operands.
    - The infix expression given above is equivalent to A B C + * D /
    - The order of evaluation of operators is always ***left-to-right***, and brackets cannot be used to change this order.
  - Prefix notation (also known as "Polish notation"): + X Y
    - Operators are written *before* their operands.
    - The expressions given above are equivalent to / * A + B C D

Arithmetic Expressions

- The purpose of an arithmetic expression is to specify an arithmetic computation.

- An implementation of such a computation must cause two actions:
  - ***Fetching*** the operands, usually from memory
  - ***Executing*** arithmetic operations on those operands

Operator Evaluation Order

- The operator precedence and associativity rules of a language dictate the order of evaluation of its operators.

Operator Evaluation Order: Precedence

- The value of an expression depends at least in part on ***the order of evaluation of the operators*** in the expression.

- Consider the following expression:

  a + b * c

  - Suppose the variables a, b, and c have the values 3, 4, and 5, respectively.
    - If evaluated ***left to right*** (the addition first and then the multiplication), the result is 35.
    - If evaluated ***right to left***, the result is 23.

Operator Evaluation Order: Precedence

- Instead of simply evaluating the operators in an expression from left to right or right to left,
  - The concept of placing operators in a hierarchy of evaluation priorities and basing the evaluation order of expressions partly on this hierarchy

- For example:
  - In mathematics, multiplication is considered to be of higher priority than addition, perhaps due to its higher level of complexity.
  - If that convention were applied in the following example expression, as would be the case in most programming languages, the multiplication would be done first.

$$a + b * c$$

Operator Evaluation Order: Precedence

- The ***operator precedence rules*** for expression evaluation partially define the order in which the operators of different precedence levels are evaluated.

- The operator precedence rules for expressions are based on the hierarchy of operator priorities, as seen by the language designer.

- The operator precedence rules of the common imperative languages are nearly all the same, because they are based on those of mathematics.
  - Exponentiation has the highest precedence (when it is provided by the language), followed by multiplication and division on the same level, followed by binary addition and subtraction on the same level.

Operator Evaluation Order: Precedence

- Many languages also include unary versions of addition and subtraction.

- ***Unary addition*** is called the ***identity operator*** because it usually has no associated operation and thus has no effect on its operand.
  - Ellis and Stroustrup (1990, p. 56), speaking about C++, call it a historical accident and correctly label it useless.

- ***Unary minus***, of course, changes the sign of its operand.
  - In Java and C#, unary minus also causes the implicit conversion of short and byte operands to int type.

Operator Evaluation Order: Precedence

- In all of the common imperative languages, the unary minus operator can appear in an expression
  - either at the beginning
  - or anywhere inside the expression, as long as it is parenthesized to prevent it from being next to another operator.

- For example,

a + (- b) * c	(legal)
a + - b * c	(illegal)

Operator Evaluation Order: Precedence

- Consider the following expressions:

```
- a / b
- a * b
- a ** b
```

- In the first two cases, the relative precedence of the unary minus operator and the binary operator is irrelevant—the order of evaluation of the two operators has no effect on the value of the expression.
- In the last case, however, it does matter.

Operator Evaluation Order: Precedence

- Of the common programming languages, only Fortran, Ruby, Visual Basic, and Ada have the exponentiation operator.
- In all four, exponentiation has higher precedence than unary minus, so

```
- A ** B
```

is equivalent to

```
-(A ** B)
```

Operator Evaluation Order: Precedence

- The precedence of the arithmetic operators of Ruby and the C-based languages are as follows:

	Ruby	C-Based Languages
Highest	**	postfix ++, --
	unary +, -	prefix ++, --, unary +, -
	*, /, %	*, /, %
Lowest	binary +, -	binary +, -

- The ** operator is exponentiation.
- The % operator takes two integer operands and yields the remainder of the first after division by the second.

Operator Evaluation Order: Associativity

- Consider the following expression:

  a - b + c - d

  - If the ***addition*** and ***subtraction*** operators have the same level of precedence, as they do in programming languages, the precedence rules say nothing about the order of evaluation of the operators in this expression.

Operator Evaluation Order: Associativity

- When an expression contains two adjacent occurrences of operators with the ***same level of precedence***, the question of which operator is evaluated first is answered by the ***associativity*** rules of the language.

- An operator can have either ***left*** or ***right associativity***, meaning that when there are two adjacent operators with the same precedence, the left operator is evaluated first or the right operator is evaluated first, respectively.

Operator Evaluation Order: Associativity

- Associativity in common languages is left to right, except that the exponentiation operator (when provided) sometimes associates right to left.
- In the Java expression

  a - b + c

  - the left operator is evaluated first.

Operator Evaluation Order: Associativity

- Exponentiation in Fortran and Ruby is right associative, so in the expression

  A ** B ** C

  - the right operator is evaluated first.

Operator Evaluation Order: Associativity

- In Ada, exponentiation is nonassociative, which means that the expression

  A ** B ** C

  - is illegal.
- Such an expression must be parenthesized to show the desired order, as in either

  (A ** B) ** C

  Or

  A ** (B ** C)

Operator Evaluation Order: Associativity

- In Visual Basic, the exponentiation operator, ^, is left associative.
- The associativity rules for a few common languages are given here:

Language	Associativity Rule
Ruby	Left: *, /, +, -
	Right: **
C-based languages	Left: *, /, %, binary +, binary -
	Right: ++, --, unary -, unary +
Ada	Left: all except **
	Nonassociative: **

Operator Evaluation Order: Associativity

- In APL, all operators have the same level of precedence.
- Thus, the order of evaluation of operators in APL expressions is determined entirely by the associativity rule, which is right to left for all operators.
- For example, in the expression

$$A \times B + C$$

  - The addition operator is evaluated first, followed by the multiplication operator
  - If A were 3, B were 4, and C were 5, then the value of this APL expression would be 27.