CMPS 3500

Programming Languages

Dr. Chengwei Lei
CEECS
California State University, Bakersfield
Chapter 7
Expressions and Assignment Statements
Chapter 7 Topics

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment
Introduction

- Expressions are the fundamental means of specifying computations in a programming language
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation
- Essence of imperative languages is dominant role of assignment statements
Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of operators, operands, parentheses, and function calls
Arithmetic Expressions: Design Issues

- Design issues for arithmetic expressions
  - Operator precedence rules?
  - Operator associativity rules?
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?
Arithmetic Expressions: Operators

- A unary operator has one operand
- A binary operator has two operands
- A ternary operator has three operands
Arithmetic Expressions: Operator Precedence Rules

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

- Typical precedence levels
  - parentheses
  - unary operators
  - ** (if the language supports it)
  - *, /
  - +, -
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., in 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.
Expressions in Ruby and Scheme

- Ruby
  - All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as methods
  - One result of this is that these operators can all be overridden by application programs

- Scheme (and Common Lisp)
  - All arithmetic and logic operations are by explicitly called subprograms
  - \( a + b \times c \) is coded as \((+ a (* b c))\)
Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:
    
    ```
    average = (count == 0)? 0 : sum / count
    ```
Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:
    \[ \text{average} = (\text{count} == 0) ? 0 : \text{sum} / \text{count} \]

- Evaluates as if written as follows:
  ```
  if (count == 0)
  average = 0
  else
  average = sum / count
  ```
Arithmetic Expressions: Operand Evaluation Order

Operand evaluation order
1. Variables: fetch the value from memory
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. The most interesting case is when an operand is a function call
### C Operator Precedence Chart

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Expression Operators</td>
<td>() [] . -&gt; expr++ expr--</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary Operators</td>
<td>* &amp; + - ! ~ ++expr --expr (typecast) sizeof</td>
<td>right-to-left</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>* / %</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ -</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt; &lt;&lt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>== !=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>^</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ternary Operator</td>
<td>?:</td>
<td>right-to-left</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment Operators</td>
<td>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
<td>=</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comma</td>
<td>,</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>
Arithmetic Expressions: Potentials for Side Effects

- **Functional side effects:** when a function changes a two-way parameter or a non-local variable

- **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;
    /* assume that fun changes its parameter */
    b = a + fun(&a);
    ```
Functional 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:** inflexibility of one-way parameters and lack of non-local references

2. Write the language definition to demand that operand evaluation order be fixed
   - **Disadvantage:** limits some compiler optimizations
   - Java requires that operands appear to be evaluated in left-to-right order
A program has the property of referential transparency if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program.

\[
\text{result1} = \frac{\text{fun}(a) + b}{\text{fun}(a) - c};
\]
\[
\text{temp} = \text{fun}(a);
\]
\[
\text{result2} = \frac{\text{temp} + b}{\text{temp} - c};
\]

If \text{fun} has no side effects, \text{result1} = \text{result2}

Otherwise, not, and referential transparency is violated.
Advantage of referential transparency

- Semantics of a program is much easier to understand if it has referential transparency

- Because they do not have variables, programs in pure functional languages are referentially transparent
  - Functions cannot have state, which would be stored in local variables
  - If a function uses an outside value, it must be a constant (there are no variables). So, the value of a function depends only on its parameters
Overloaded Operators

- Use of an operator for more than one purpose is called operator overloading
- Some are common (e.g., + for int and float)
- Some are 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
Overloaded Operators (continued)

- C++, C#, and F# allow user-defined overloaded operators
  - When sensibly used, such operators can be an aid to readability (avoid method calls, expressions appear natural)
  - Potential problems:
    - Users can define nonsense operations
    - Readability may suffer, even when the operators make sense