National University
Of Computer and Emerging Sciences

Theory of Programming Languages

Expressions and Assignment Statements

Sajid Anwer

Department of Computer Science,
FAST-NUCES, CFD Campus

Chapter Outline

- 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 & Precedence Rules

- 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
- Typical precedence levels
  - » parentheses
  - » unary operators
  - » ** (if the language supports it)
  - » *, /
  - » +, -

Arithmetic Expressions: Operator Associativity Rule

- 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 *overriden* with *parentheses*

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - » C-based languages (e.g., C, C++)
  - » An example:

```
average = (count == 0)? 0 : sum / 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

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 or 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

Referential Transparency

- 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

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

- If *fun has no side effects*, result1 = 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, *how?*

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, *how*?

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`**

Type Conversions: Mixed Mode

- A *mixed-mode expression* is one that has operands of different types

- A *coercion* is an implicit type conversion

- Disadvantage of coercions:
  - » They decrease in the type error detection ability of the compiler

```
int a;
float b, c, d;
. . .
d = b * a;
```

- In most languages, all numeric types are coerced in expressions, using widening conversions

Explicit Type Conversions

- Called *casting* in C-based languages
- Examples
  - » C: `(int)angle`
  - » F#: `float(sum)`

Errors in Expressions

- Causes
  - » Inherent limitations of arithmetic; e.g., division by zero
  - » Limitations of computer arithmetic; e.g. overflow

- Often *ignored* by the run-time system

Reading Activity

- Subsections 7.5 to 7.8.