Programming Languages:

Lecture 7

Chapter 7: Expressions and Assignment Statements

Jinwoo Kim
jwkim@jjay.cuny.edu

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
  - mode mixing 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
  - E.g. a + b * c (when a = 3, b = 4, c = 5)

- 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
  - E.g. a – b + c - d
- Typical associativity rules
  - Left to right, except **, which is right to left
    - **E.g. a ** b ** c**
      - **Fortran and Ada handle above expression differently**
- APL is different
  - all operators have equal precedence and all operators associate right to left
- Precedence and associativity rules can be overridden with parentheses

Arithmetic Expressions: Parentheses

- Programmers can alter the precedence and associativity rules by placing parentheses in expressions
  - E.g. (a + b) * c

- Languages that allow parentheses in arithmetic expressions could dispense with all precedence rules and simply associate all operators either left to right or right to left
  - The programmer can specify desired order of evaluation with parentheses
  - Advantage: Simple, now programmer does not need to remember any precedence or associative rules
    - **APL follows this approach**
      - **E.g. A x B + C**
  - Disadvantage: Can makes writing expressions more tedious which can also yields readability problems

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - Expression1 ? Expression2 : expression3
    - **C-based languages (e.g., C, C++)**
  - An example:

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

  - Evaluates as if written like

```
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

- Operand evaluation order becomes interesting when it does have *side effects*

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., function changes a global variable:

```
int a = 10;

int fun1(){
   a = 20;
  return 3;
}

int fun2(){
   a = a + fun1();
}

void main(){
   fun2();
}
```

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 two-way parameters and non-local references**
2. Write the language definition to demand that operand evaluation order be fixed
   - Disadvantage**: limits some compiler optimizations**

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
  - Can be avoided by introduction of new symbols
    - **e.g., Pascal's `div` for integer division**
      - **avg := sum / count (floating point division in Pascal)**
      - **avg = sum / count (integer division in C or C++ if sum and count are integer type)**

Overloaded Operators (continued)

- C++ and Ada allow user-defined overloaded operators
  - Exceptions: . ::

- Potential problems:
  - Users can define nonsense operations
    - **E.g. User can define + to multiply**
  - Readability may suffer, even when the operators make sense
    - **E.g. Seeing an * operator in a program, the reader must find both the types of the operands and the definition of the operators to determine its meaning**

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`
  - Usually safe but may result in certain problem

Implicit Type Conversions

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

- A *coercion* is an implicit type conversion
  - Initiated by compiler
  - Gives flexibility to the language

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

Implicit Type Conversions (Continued)

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

- In Ada, there are virtually no coercions in expressions
  - Does not usually allow operand type mixing

Explicit Type Conversions

- Explicit Type Conversions
  - Type conversion explicitly requested by programmer

- Called *casting* in C-based language

- Examples
  - `C: (int) angle`
  - `Ada: Float (sum)`

Note that Ada’s syntax is similar to function calls

Type Conversions: Errors in Expressions

- Causes
  - Inherent limitations of arithmetic
    - **e.g., division by zero**
  - Limitations of computer arithmetic
    - **e.g. overflow or underflow**

- Often ignored by the run-time system or sometimes calls error handling routine called "exceptions"

Relational and Boolean Expressions

- Relational Expressions
  - Use relational operators and operands of various types
    - **Typical types for relational operators: numeric, string, ordinal types**
  - Evaluate to some Boolean representation
  - Operator symbols used vary somewhat among languages (`!=`, `/=`, `.NE.` `<>`)

Relational and Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Example operators

FORTRAN 77	**FORTRAN 90**	**C**	**Ada**
`.AND.`	`and`	`&&`	`and`
`.OR.`	`or`	`||`	`or`
`.NOT.`	`not`	`!`	`not`
			`xor`

Relational and Boolean Expressions: No Boolean Type in C

- 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:
  - Left operator is evaluated, producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., `c`)

Relational and Boolean Expressions: Operator Precedence

- Precedence of C-based operators

```
postfix ++, --
unary +, -, prefix ++, --, !
*,/,%
binary +, -
<, >, <=, >=
==, !=
&&
||
```

Short Circuit Evaluation

- An expression in which the result is determined without evaluating all of the operands and/or operators

- Example: `(13*a) * (b/13-1)`

  If `a` is zero, there is no need to evaluate `(b/13-1)`

  But unlike *Boolean expression*, it is not easy to detect shortcut in *arithmetic expression*

- Better Example: `(a >= 0) && (b < 10)`

  This shortcut can be easily discovered during execution

- Problem with non-short-circuit evaluation

```
index = 0;
while (index <= length) && (LIST[index] != value)
      index++;
```

  – When `index=length, LIST [index]` will cause an indexing problem (assuming `LIST` has `length -1` elements)

Short Circuit Evaluation (continued)

- 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`)

- Short-circuit evaluation exposes the potential problem of side effects in expressions
  - e.g. `(a > b) || (b++ / 3)`

Assignment Statements

- The general syntax

  <target_var> <assign_operator> <expression>

- The assignment operator

  `=` FORTRAN, BASIC, PL/I, C, C++, Java

  `:=` ALGOLs, Pascal, Ada

- = can be bad when it is overloaded for the relational operator for equality

Assignment Statements: Conditional Targets

- Conditional targets (C, C++, and Java)

```
(flag)? total : subtotal = 0
```

Which is equivalent to

```
if (flag)
   total = 0
else
   subtotal = 0
```

Assignment Statements: Compound Operators

- A shorthand method of specifying a commonly needed form of assignment
  - Destination variable also appear as the first operand in the expression on the right side

- Introduced in ALGOL; adopted by C

- Example

```
a = a + b
```

is written as

```
a += b
```

Assignment Statements: Unary Assignment Operators

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment

- Examples

  `sum = ++count` (`count` incremented, assigned to `sum`)

  `sum = count++` (`count` assigned to `sum`, incremented)

  `count++` (`count` incremented)

  `-count++` (`count` incremented then negated)

Assignment as an Expression

- In C, C++, and Java, the assignment statement produces a result and can be used as operands

- An example:

```
while ((ch = getchar())!= EOF){…}
```

`ch = getchar()` is carried out; the result (assigned to `ch`) is used as a conditional value for the `while` statement

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode, for example

```
int a, b;
float c;
c = a / b;
```

- In Fortran and C-based languages, coercion is freely allowed
  - E.g., int to float or float to int
- In C# and Java, only widening assignment coercions are done
- In Ada, there is no assignment coercion

Summary

- Expressions
- Operator precedence and associativity
- Operator overloading
- Mixed-type expressions
- Various forms of assignment

Homework #3 (part 3)

- Problem Solving (P. 345 of class textbook)
  - 8,13
- Due date: One week from assigned date
  - Please hand in printed (typed) form
    - **I do not accept any handwritten assignment**
    - **Exception: pictures**