Lecture 5: Expressions and Assignment Statements

Chapter 7

Lecture 5 (Chapter 7) Topics

- Introduction
- Arithmetic Expressions
- Conditional Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignments
- 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
  - A unary operator has one operand
  - A binary operator has two operands
  - A ternary operator has three operands
- In most languages, binary operators are infix, except in Scheme and LISP, in which they are prefix; Perl also has some prefix binary operators
- Most unary operators are prefix, but the ++ and -- operators in C-based languages can be either prefix or postfix

  ++j (prefix) j++ (postfix)

Infix vs. Prefix

Lisp Expression (prefix) (- a (+ (* b c) (/ d 3)))

⇒ Java (infix) a - (b* c + d/3)

Can you write the quadratic formula in both prefix and infix respectively?

quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

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?

1. Run under https://www.onlinegdb.com/online_c++_compiler
2. Rewrite the code into Java and run

```
#include <iostream>

using namespace std;

int main()
{
    int x = 1, y =2;
    int result = y + (x=++y);
    cout <<"result= "<< result;
    return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    int x = 1, y =2;
    int result = (y+1) + (x=++y);
    cout <<"result= "<< result;
    return 0;
}
```

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)
  - *, /
  - +, -

- Question: is Java's operator precedence (& associativity) rules identical to that of your team language? If different, give an expression that would be interpreted differently by these two languages.

- Java's rule

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Order of evaluation in Java: a < b == c > d

(1)
(2)
(3)

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 ** (power/exponent) right to left

    $4 ** 3 ** 2 = 2^x \quad x = ?$
- APL is different; all operators have equal precedence and all operators associate right to left
- Precedence and associativity rules can be overriden with parentheses

Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)

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

  - Evaluates as if written as follows:

```
if (count == 0)
   average = 0;
else
   average = sum /count;
```

- Question: What other language(s) support conditional expression? Write the above example (conditional expression) in that language's syntax.

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

```
a = 10;
b = a + fun(&a); // assume that fun changes its parameter
          //what will be the value a now? if modify code to
b = fun(&a) + a;          //x + y same as y + x?
```

```
int fun (int *p) { *p = *p + 1; return *p; }

a = 10;
b1 = (a + 1) + fun(&a);

a = 10;
b2 = fun(&a) + (a + 1);
```

b1 == b2 ?

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

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 result1 == result2, no side effect; otherwise referential transparency is violated

- Advantage of referential transparency
  - Semantics of a program is much easier to understand if it has referential transparency
- Programs in pure functional languages are referentially transparent because no local variables

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++)

    `c = a * b;` vs. `c = a * *pb;`
  - Disadvantages:
    - Loss of compiler error detection (omission of an operand should be a detectable error)
    - Some loss of readability
- C++, C#, and F# allow user-defined overloaded operators
  - When sensibly used, aid to readability and writability (avoid method calls, expressions appear natural)
  - Potential problems:
    - Users can define nonsense operations
  - More to discuss in OOP

Type Conversions and Mixed-Mode

- *narrowing conversion:* converts an object to a type that cannot include all of the values of the original type

  e.g., convert float to int

- *widening conversion:* an object is converted to a type that can include at least approximations to all of the values of the original type

  e.g., convert int to float

- *mixed-mode expression:* has operands of different types

  e.g., 3.5 + 35

Coercion and Explicit Type Conversions

- A *coercion* is an implicit type conversion
  - Disadvantage: decrease in the type error detection ability of the compiler
- In most languages all numeric types are coerced in expressions, using widening conversions
  - Exception: ML and F# no coercion, explicit type conversion required for mixed-mode operations
- Explicit type conversions
  - Called *casting* in C-based languages

```
(int)angle          //C
float(sum) + 3.0    //F#
```

//F# no coercion-- if sum is integer, sum + 3.0 type error

Relational Expressions

- Relational Expressions
  - Use relational operators and operands of various types
  - Evaluate to some Boolean representation
  - Operator symbols used vary somewhat among languages

    `!=, /=, ~=, .NE., <>, …`
- JavaScript and PHP have two additional relational operator, `===` and `!==`
  - Similar to their cousins, `==` and `!=`, except that they do not coerce their operands
  - Ruby uses `==` for equality relation operator that uses coercions and `eql?` for those that do not
  - Question: Pros and Cons for introducing `===` and `!==`?

Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Operator symbols vary somewhat: and, &&, …
- Boolean type
  - Boolean or bool, … only two values: true, false
    - TRUE, True, true, … (case sensitive)
  - Some languages have 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**
    - A legal expression, but the result is not what you might expect:
    - Example: -3 < -2 < -1

      -3 < -2 produces 1 (true)

      1 < -1 produces 0 (false)
  - Question: what is the result of Python expression -3 < -2 < -1?

Short Circuit Evaluation

- Short-circuit evaluation of an expression
  - The result of the expression is determined without evaluating all of the operands and/or operators
  - Non-short-circuit evaluation: strict evaluation
- Short circuit evaluation example

  (13 * a) * (13 / b – 1)

  If a is zero, there is no need to evaluate (b /13 – 1)
- Problem with strict evaluation

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

  - When index=length, LIST[index] will cause an indexing problem (assuming LIST is length – 1 long)
- Short-circuit evaluation: more efficient, better reliability

Short Circuit Evaluation: Language Support

- Early languages does not support short-circuit evaluation
  - Early version of FORTRAN, PASCAL, …
- Later on both short-circuit and strict evaluations are supported, with strict evaluation as default
  - e.g. Ada and, or (strict), and then, or else (short circuit)
- Then, both version supported with short circuit as default/recommended
  - 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 |)
- Some newer language use short-circuit only
  - Ruby, Perl, ML, F#, and Python
- 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, the C-based languages, …

  := Ada, Pascal, …

  Problem: confusion between =, == and :=

```
if a = b then sum := a; else sum := 1;   //Pascal
if (a=b) { … } else { … }
                //common mistake in C-based languages
//What happens in C++ and Java respectively?
int a = 5, b = 4;
if (a=b) { … “good” …} else { … “okay” …}
```

Conditional Targets

- Conditional targets (Perl)

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

Which is equivalent to

```
if ($flag){
    $total = 0
} else {
    $subtotal = 0
}
```

Question: any other language(s) support assignment with conditional target? If yes, name the language and give a line of code as example.

Compound/Augmented and Unary Assignments

- Shorthand methods of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by the C-based languages
- Compound/augmented assignment operators

  Example: a = a + b can be written as a += b
- Unary assignment operators combine increment and decrement operations with assignment, for example,

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

  sum = count++ (count assigned to sum, then incremented

  count++ (count incremented)

  -count++ (count incremented then negated)

Questions:

(1) pros and cons of using such operators, e.g. p[j++] = q[++j]++;

(2) Python vs. Java with regard to compound and unary assignments

Assignment as an Expression

- In the C-based languages, Perl, and JavaScript, the assignment statement produces a result and can be used as an operand

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

  ch = getchar() is carried out; the result (assigned to ch) is used as a conditional value for the while statement
- Advantage: writability
- Disadvantage: (another kind of) expression side effect

Multiple Assignments

- Perl and Ruby allow multiple-target multiple-source assignments

```
($first, $second, $third) = (20, 30, 40);
```

Also, the following is legal and performs an interchange:

```
($first, $second) = ($second, $first); //swap!
```

Question:

(1) Any other language(s) support multiple assignments?

(2) Does the above ($first, $second) relates to a tuple?

(3) Would the syntax such as first, second = second, first; better in readability?

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode

  Example: C++

```
int num = 3;
double sum;
sum = num;  //okay
num = sum;  //any problem?
```

  - In Fortran, C, Perl, and C++, any numeric type value can be assigned to any numeric type variable
  - In Java and C#, only widening assignment coercions are done
  - In Ada, there is no assignment coercion

Summary

- Expressions
  - Arithmetic, conditional, relational, Boolean, …
- Operator precedence and associativity
  - Varies among languages
- Operator overloading
- Mixed-type expressions
- Various forms of assignment
- Common errors in expressions
  - Inherent limitations of arithmetic, e.g., division by zero
  - Limitations of computer arithmetic, e.g. overflow
  - Not discussed here but be aware