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)
  – *, /
  – +, -

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

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 overriden by application programs

- Scheme (and Common LISP)
  - All arithmetic and logic operations are by explicitly called subprograms
  - `a + b * 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
```

  – 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 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 `fun` has no side effects, `result1 = result2`

Otherwise, not, and referential transparency is violated

Referential Transparency (continued)

- 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

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
- In most languages, all numeric types are coerced in expressions, using widening conversions
- In Ada, there are virtually no coercions in expressions
- In ML and F#, there are no coercions in expressions

Explicit Type Conversions

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

Note that F#'s syntax is similar to that of function calls

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

Relational and Boolean 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

Relational and Boolean Expressions

Relational and Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Example operators
- C89 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`**)

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)`
- 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` is `length - 1` long)

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 `|`)
- All logic operators in Ruby, Perl, ML, F#, and Python are short-circuit evaluated
- 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, the C-based languages

  := Ada

- = can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use == as the relational operator)

Assignment Statements: Conditional Targets

- Conditional targets (Perl)

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

Which is equivalent to

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

Assignment Statements: Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by C and the C-based languaes
  - Example

```
a = a + b
```

can be 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, then assigned to `sum`)

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

`count++` (`count` incremented)

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

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

- Disadvantage: another kind of expression side effect

Multiple Assignments

- Perl, Ruby, and Lua 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);
```

Assignment in Functional Languages

- Identifiers in functional languages are only names of values
- ML
  – Names are bound to values with **`val`**

    ```
    val fruit = apples + oranges;
    ```

  - If another val for fruit follows, it is a new and different name
- F#
  – F#'s **`let`** is like ML's **`val`**, except **`let`** also creates a new scope

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode
- 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
- Operator precedence and associativity
- Operator overloading
- Mixed-type expressions
- Various forms of assignment