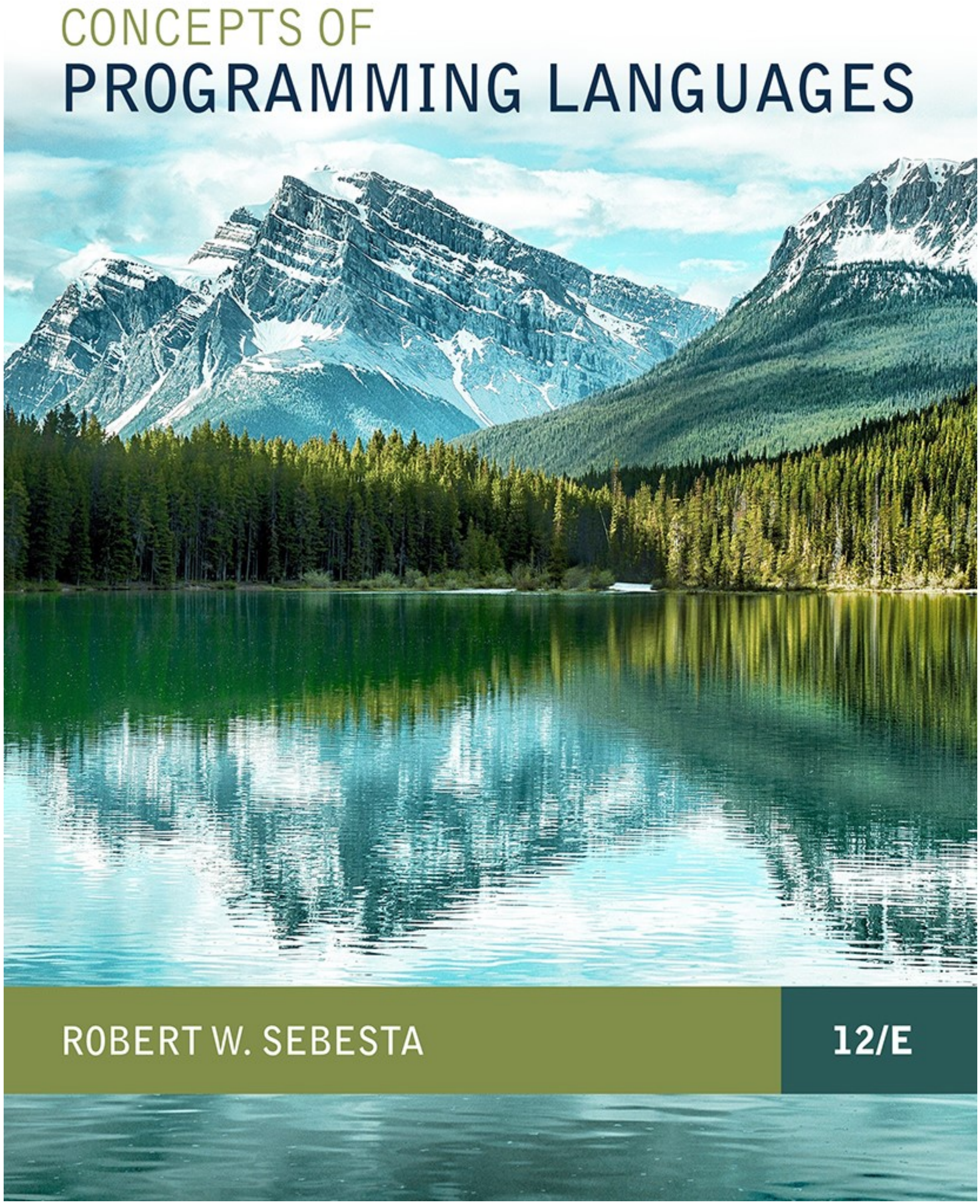
Chapter 7

Expressions and Assignment Statements



ISBN 0-321-49362-1

Chapter 7 Topics

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

Introduction

- Expression:
  - the fundamental means of specifying computations in a programming language
- 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
  - one of the motivations for the development of the first programming languages
- Arithmetic expression
  - consist of operators, operands, parentheses, and function calls
- e.g.,
  - In Lisp: (+ a (* b c)) // a + b*c
  - In Java : (x+y)/100

Arithmetic Expressions: Design Issues

- Design issues for arithmetic expressions
  - Operator:
    - precedence rules
    - associativity rules
    - overloading
  - Operand:
    - Evaluation order?
    - Evaluation side effects?
  - Type mixing in expressions?

Arithmetic Expressions: Operators

- Unary operator has one operand
  - E.g., Java: count++;
- Binary operator has two operands
  - E.g., Java: HW + exam;
- Ternary operator has three operands
  - E.g., Java: (a) ? b : c; // conditional operator

Arithmetic Expressions: Operator Precedence Rules

- The *operator precedence rules*
  - define the order in which "adjacent" operators of different precedence levels are evaluated
- Typical precedence levels
  - parentheses
  - unary operators
  - Exponentiation ** (Ruby, Ada: $2**3 = 2^3 = 8$)
  - *, /
  - +, -

Arithmetic Expressions: Operator Associativity Rule

- Operator associativity rules:
  - define the how operators with the same precedence level are evaluated
- Typical associativity rules:
  - Left to right:
    - E.g., Java: a + b – c + d + e; // left to right
  - **, which is right to left:
    - E.g., Ruby: 2 ** 2 ** 3 = 256 // right to left
- APL is different
  - all operators have equal precedence
  - all operators associate right to left
    - E.g., 3 X 4 + 5 = 27

Expressions in Ruby and Scheme

- Ruby
  - operators are implemented as methods
    - arithmetic, relational, assignment, array indexing, shifts, and bit-wise logic
  - operators can be overridden 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++)
  - E.g.,:

```
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*
  - Variables:
    - fetch the value from memory
  - Constants:
    - fetch from memory; sometimes is in the machine language instruction
  - Parenthesized expressions:
    - evaluate all operands and operators first

Arithmetic Expressions: Potentials for Side Effects

- Functional side effects:
  - 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 (e.g., g) of the expression;
  - e.g., in C++

```
int g=10;
int fun() { g=20;  return g;}
int main(int  arg, char* args[]) {
     printf("%d\n", fun()+g);       // 40
     printf("%d\n", g+fun());       // 30
     return 0;}
```

Functional Side Effects

- Two possible solutions to the problem
  1. 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. 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

- 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 as they do not have variables
  - 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

- *operator overloading:*
  - use of an operator for more than one purpose
- 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)
  - loss of readability

Overloaded Operators

- 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
      - E.g., a = b * c;
        a.assign(b.mul(c));

Type Conversions

- *Narrowing conversion:*
  - an object is converted to a type that cannot include all of the values of the original type
  - e.g., `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., `int` to `float`

Type Conversions: Mixed Mode

- *Mixed-mode expression*:
  - has operands of different types
- *Coercion:* an implicit type conversion
  - Disadvantage of coercions:
  - decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using **widening** conversions
  - e.g., in Java
  - int a; float b, c, d; d = b * a; // convert int a to float
- In ML and F#
  - there are no coercions in expressions

Explicit Type Conversions

- Called *casting*
- Examples
  - C: `(int)angle`
  - F#: `float(sum)`
  - Java: `Vehicle v; Sedan s = (Sedan) v;`

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

Errors in Expressions

- Causes
  - limitations of arithmetic

    e.g., division by zero
  - Limitations of computer arithmetic

    e.g. overflow (positive or negative)

Relational Expressions

- Relational Expressions
  - Use relational operators and operands of various types, e.g., a > b
  - Evaluate to some Boolean representation
  - Operator symbols used vary somewhat among languages (`!=`, `/=`, `~=`, `.NE.`, `<>`, `#`)
- JavaScript and PHP
  - 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

Boolean Expressions

- Boolean Expressions consist of:
  - Boolean variable, Boolean constant, relational expression and Boolean operators (e.g., AND, OR, NOT)
- Boolean operators take:
  - Boolean operands--Boolean variables or relational expression
- C89 has no Boolean type
  - 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:
  - Left operator is evaluated (a and b), producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., `c`)

Short Circuit Evaluation

- Result of an expression is determined without evaluating all of the operands and/or operators
  - E.g.,: `(13 * a) * (b / 13 - 1)`
    If `a` is zero, 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` length is `length - 1`)

Short Circuit Evaluation

- 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
- Short-circuit evaluation exposes the potential problem of side effects in expressions

  e.g. `(a > b) || (b++ / 3)`

`Note: b is changed only when a <= b, not every time this expression is evaluated.`

Assignment Statements

- General syntax:

  `<target_var> <assign_operator> <expression>`
- Assignment operator

  = Fortran, BASIC, 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 (in 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 languages
  - E.g.,

```
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
  - E.g.,

  `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 (MetaLanguage)
  - Names are bound to values with **`val`**

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

  - If another val for **fruit** follows, it is a new, different identifier.
- F#
  - Binding expression to define values for one or more names

    let i = 1

    let i, j, k = (1, 2, 3)

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--coercion freely applied
- In Java and C#
  - only widening assignment coercions are allowed
- In Ada,
  - there is no assignment coercion