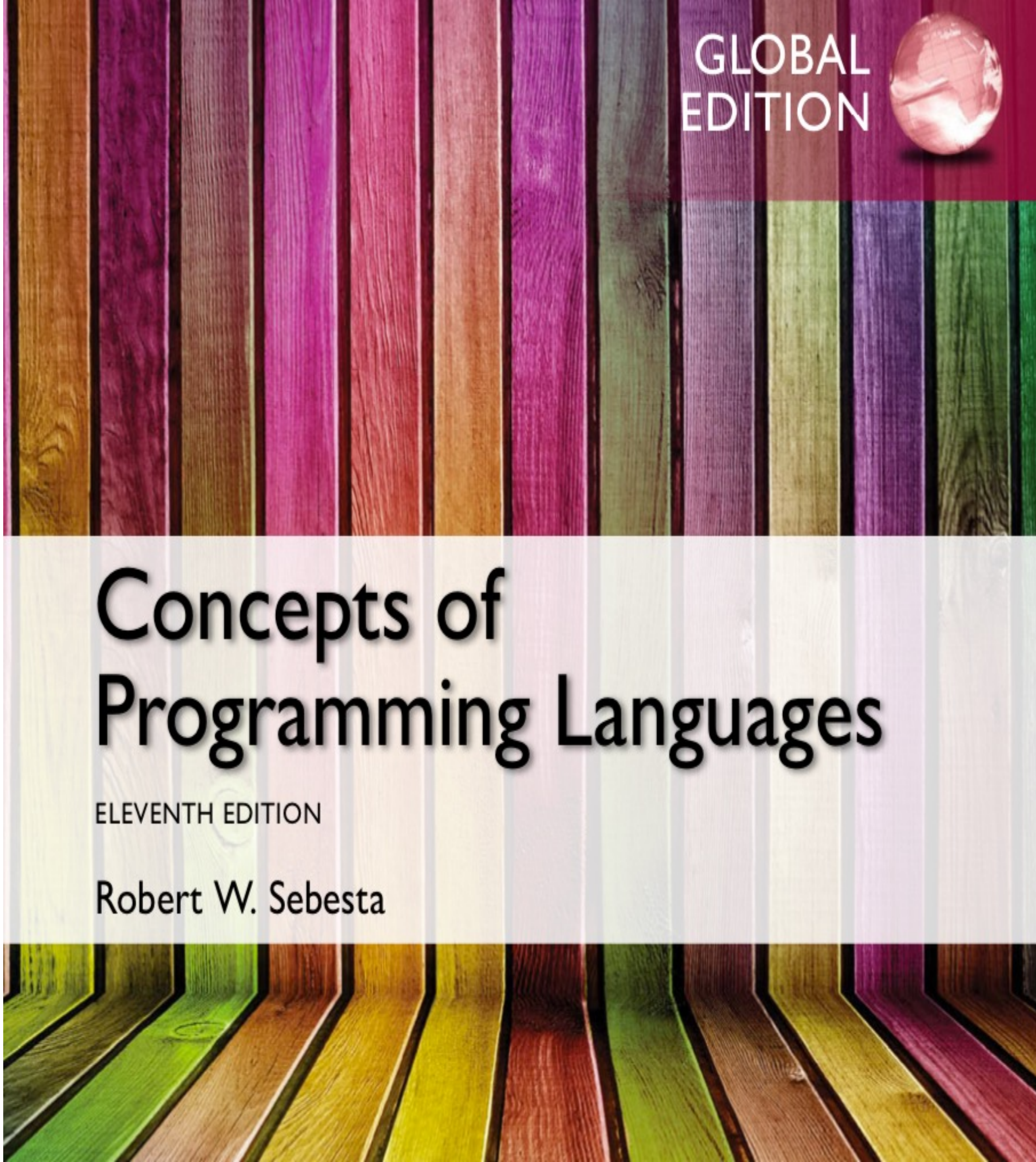
Chapter 6

Expression and Assignment Statement



Introduction

- Introduction
- Arithmetic Operation
- Overloaded Expression
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluations
- Assignment Statement
- Mixed-Mode Assignment

Introduction

- Expressions are the fundamental means of specifying computations in a programming language. y = c + 5 / x
- To understand expression evaluation, it is necessary to be familiar with the orders of operator and operand evaluation.
- The essence of the imperative programming languages is the dominant role of assignment statements.

Arithmetic Expression

- Arithmetic Expression was one of the motivation in the creation of the earlier programming languages.
- Arithmetic expressions consist of operators, operands, parentheses, and function calls.
- Example y = c + x – (d - z) / s

Arithmetic Expression

- Design Issue for Arithmetic Expression
  - Operator precedence rules.
  - Operator associativity rules
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?

Arithmetic Expression

- Operators:
  - **Unary Operator**, meaning it has a single operand.
  - i++, i--
  - **Binary Operator**, meaning it has two operands.
  - y + x , z / 4
  - **Ternary,** meaning it has three operands.
  - M = (n1 > n2) ? n1 : n2 ;
  -

Arithmetic Expression

- Operator Precedence:
  - The operator precedence rules for expression evaluation define the order in which the “adjacent” operators of different precedence levels are evaluated.
  - Typical Precedence level
  - Parentheses ( )
  - Unary operator ++ , --
  - **(if language supports)
  - /,*
  - + ,-
  - y = x + f / 6 – (3 - g)

Arithmetic Expression

- Operator Associativity rule:
  - The operator associativity rules for expression evaluation define the order in which the "adjacent" operators of same precedence levels are evaluated
  - Y = C * X / Y * F / G
- Typical Associativity rule
  - Left to right, expect ** which is right to left
  - x = 15
  - y = 4
  - x **y = 502625
- Sometimes unary operator associate from right to left.

Arithmetic Expression

- Ruby:
  - Ruby is a pure object-oriented language, which means, among other things, that every data value, including literals, is an object
  - all of the arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bitwise logic operators, are implemented as methods
  - Result of the implementation of operators as methods is that they can be overridden by application programs
- Lisp
  - all arithmetic and logic operations in Lisp are performed by subprograms
  - a + b * c is coded as (+ a (* b c))

Arithmetic Expression

- Conditional Expression:
  - In C based languages
  - average = (count == 0) ? 0 : sum / count;

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

Operand Evaluation Order

- Variable: fetch the variable from the memory
- Constants: some time fetch from memory, or may be in machine language instructions.
- Parenthesized Expression: evaluate all operands and operators first.
- The most interesting case is when an operand is a function call:

```
int x = 7, y = 8
function(x,y)
```

Side Effects

- A side effect of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable.
- Problem with functional side effect.
  - When function referenced in an expression alters another operand of the expression e.g. for a parameter change.

    a = 10

    b = a + func(&a);

Side Effects

- int a = 5;
- int fun1() {
-         a = 17;
-         return 3;

  } /* end of fun1 */
- void main() {

  a = a + fun1();
- } /* end of main */

Side Effect

- Two solutions:
- Write language definition to disallow functional side effects.
  - No two- way parameter in function
  - No local reference in function
  - Disadvantage: inflexibility of one-way parameters and

  lack of non reference variable
- Write the language to demand that operand evaluation be fixed:
  - Disadvantage: limit some compiler optimizations
  - Java requires that operands appears to be evaluated in left to right order.

Side Effect

- Two solutions:
- Write language definition to disallow functional side effects.
  - No two- way parameter in function
  - No local reference in function
  - Disadvantage: inflexibility of one-way parameters and

  lack of non reference variable
- Write the language to demand that operand evaluation be fixed:
  - Disadvantage: limit some compiler optimizations
  - Java requires that operands appears to be evaluated in left to right order.

Operator Overloading

- 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++, / for int and float in Java
  - Loss of compiler error detection
  - Missing operand should be a detectable error
  - Some loss of readability
  - Can be avoided by introduction of new symbols

User-defined Overloaded Operators

- C++ and Ada allow user-defined overloaded operators
- Problems:
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense.

Type Conversions

- Narrowing conversion
  - converts to a “smaller” type (type has fewer values)
  - e.g., float to int
  - 3.99 to 4
- Widening conversion
  - converts to a type that includes all values of the original type
  - or at least an approximation of each
  - e.g., int to float
  - 4 to 4.0f

Type Conversions(Implicit)

- Mixed-mode expression
  - Operands of different types (12+ “a”+ 11.056)
- Coercion
  - An implicit type conversion
  - Double x , x = 3 , int = 3.0
- Disadvantage
  - Decreases the type error detection ability of the compiler
- In most languages, widening conversions of numeric types in expressions can be coerced
- In Ada, there are virtually no coercions in expressions

Type Conversions(Explicit)

- In C, C++, Ada, Java called casts
- E.g., Ada
  - FLOAT (INDEX)   --INDEX is INTEGER type
  - converts to floating point
- E.g., Java
  - float speed = 45.5;
  - (int) speed;   /* =45; cuts off fractional part*/

Errors in Expressions

- Inherent properties of mathematical functions
  - e.g. division by zero, infinity
- Approximate representations
  - Fractions (e.g. 2/3, 0.1) and irrational numbers like π and e
  - Approximate huge integers with floating point
- Limitations of computer arithmetic
  - e.g. overflow, underflow
- If ignored by the run-time system (may even be undetectable) can lead to crashes, erroneous output, unpredictable behavior
- Less of a problem in some languages!
  - E.g. exact fractions and huge integers in Lisp prevent errors of type 2 & 3

Relational Operators, Boolean Expressions

- Boolean data type
  - 2 values
  - True
  - False

- Boolean expression
  - Has relational operators and operands of various types
  - Evaluates to a Boolean value
  - Operator symbols vary among languages
    - e.g.not equal
    - !=
    - /=
    - .NE.
    - <>
    - #

Relational Operators, Boolean Expressions

- Operands are Boolean
- Result is Boolean
  - Boolean operator comparison

F77	FORTRAN 90	C	Ada	Lisp
.AND.	and	&&	and	and
.OR.	or	||	or	or
.NOT.	not	!	not	not
			xor	xor

Odd Boolean Expressions in C

- C (until very recently) had no Boolean type
  - used int 0 for false, and 1 or nonzero for true

- One odd characteristic of C's expressions:

  x < y < z

  - Is a legal expression, but
  - the result is not what you might expect! - I.e.(x<y)&(y<z)
  - What does it do?
  - Hint: C is left associative, what is z compared to

Short Circuit Evaluation

- A short-circuit evaluation of an expression is one in which the result is determined without evaluating all of the operands and/or operators. For example, the value of the arithmetic expression . (13 * a) * (b / 13 - 1)
- Problem
  - table look-up
  - for (i = 1; i < a.length) && (a [i] != x); i++) {}
- Problem: reading from a file until eof
- Short-circuit evaluation has the problem of side effects
  - e.g. (a > b) || (b++ / 3) vs. a > b) || (++b / 3)

Short Circuit Evaluation in PLs

- C, C++, Java
  - Provide short-circuit Boolean operators && and||
  - As well as operators that are not short circuit: & and|
  - why both?
- Ada
  - More operators, programmer can specify either
  - Not short circuit using and, or
  - Short-circuit using and then, or else
- FORTRAN 77
  - short circuit, any side-affected variables must be set to undefined

Assignment Statements

- Assignment operator syntax
  - = FORTRAN, BASIC, PL/I, C, C++, Java
  - := ALGOLs, Pascal, Ada
  - setf/setq in Lisp

- Very bad if assignment = overloaded as relational =
  - e.g. in PL/I: A = B = C;

- Note difference from C's
  - ==
  - A common C error using = when it should be ==

Complex Assignment Statements

- Multiple targets (PL/I)
  - A, B = 10
- Compound assignment operators in C, C++, Java
  - sum += next;
- Conditional targets in C, C++, Java
  - (first == true) ? total : subtotal = 0
- Unary assignment operators in C, C++, Java
  - a++;
- C, C++, and Java treat = as an arithmetic binary operator
  - a = b * (c = d * 2 + 1) + 1

Assignment Statement as an Expression

- In C, C++, Java
  - Assignment statements produce results
  - So, they can be used as operands in expressions
  - while ((ch = getchar()) != EOF){…}
- Disadvantages
  - Another kind of expression side effect
  - Readability

Mixed-Mode Assignment

- FORTRAN, C, C++
  - any numeric value can be assigned to any numeric variable
  - conversion is automatic
- Pascal
  - integers can be assigned to reals, but
  - reals cannot be assigned to integers
  - must specify truncate or round
- Java
  - only widening assignment coercions are done
- Ada
  - no assignment coercion