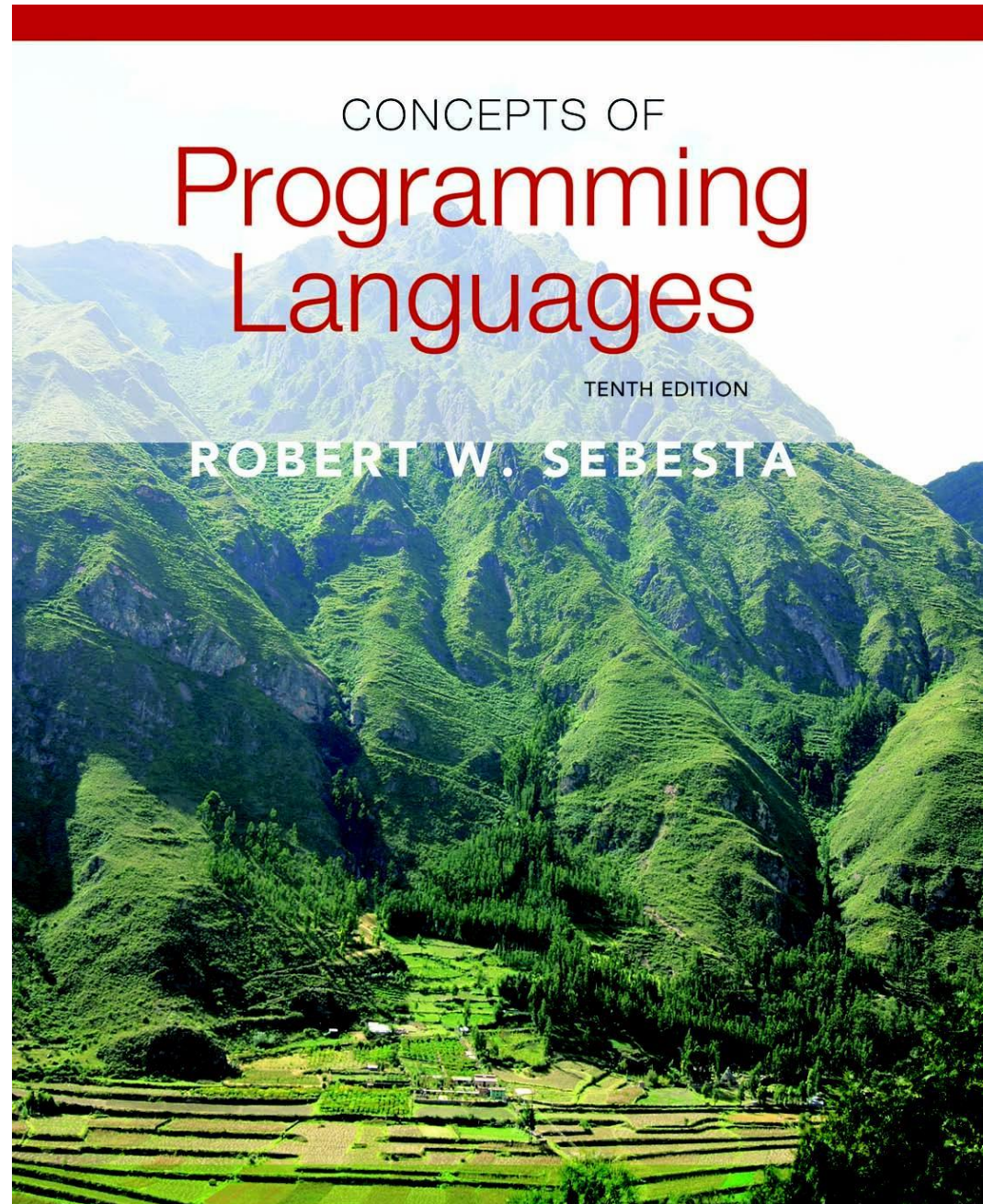


Chapter 7

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 primary goals of the first high-level programming languages
- Arithmetic expressions consist of **operators, operands, parentheses, and function calls**

$$15 * (a + b) / \log(x)$$

Arithmetic Expressions: Design Issues

- **Design issues for arithmetic expressions**
 - Operator precedence rules?
 - Operator associativity rules?
 - Order of operand evaluation?
 - Operand evaluation side effects?
 - User-defined operator overloading?
 - Type mixing in expressions?

Arithmetic Expressions: Operators

- A **unary** operator has one operand

-6

- A **binary** operator has two operands

12 * 3

infix

- A **ternary** operator has three operands (**?:**)

Condition **?** Condition is T **:** Condition is F

(i%2) **?** "odd" **:** "even"

True?

False?

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

(Highest)

– **parentheses**

– **Postfix ++, --**

– **Prefix ++, --**

– **Unary +, -**

– ****** (if the language supports it): Fortran, Ruby, VB, Ada have the exponentiation operator (지수 연산자)

– ***, /, %**

(Lowest)

– **Binary +, -**

(ref) p. 341, Ruby and C-based Languages)

- parentheses: **()**

- braces or curly brackets: **{ }**,

- square brackets **[]**

※ left brace : **{** , right parenthesis **)**

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)

Java expression : **a - b + c ;**
// the left operator (-) is evaluated first

- Precedence and associativity rules can be overridden with parentheses (p. 343 ~344)

(A + B) + (C + D) ; // to avoid overflow

Expressions in Ruby and Scheme (skip~)

- 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 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++, Java, Javascript, Perl, Ruby)
- Form: **expr_1? Expr_2: expr_3**
- 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**
 - **Variables**: fetch the value from memory
 - **Constants**: sometimes a fetch from memory; sometimes a constant may be part of the machine language instruction (not require a memory fetch)
 -  **괄호 Parenthesized expressions**: evaluate all operands and operators first
 - **The most interesting case** is when the evaluation of an operand does have **side effects**.
 - 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

one-way == 함수에 인자를 넘겨주기만 하는것(return void)
two-way == 함수에서 인자를 다시 받는것 ex) c = f(x,y);

- Problem with functional side effects:
 - When **a function** referenced in an expression **alters another operand** of the expression;
e.g., for a parameter change (p. 346)

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**

Operand Evaluation Order (functional side effects)

// Consider following situation:
// **fun** return **10** and **changes of the value of**
// **its parameter** to **20**.

```
a = 10;  
b = a + fun(a);
```

Two Results

1st : left-to-right order evaluation → $10 + 10 = 20$

2nd: right-to-left order evaluation → $20 + 10 = 30$

Operand Evaluation Order (in C)

[소스 1]

```
01 #include <stdio.h>
02 int x = 10; // global variable
03 int func(void)
04 {
05     x = 20;
06     return 30;
07 }
08 int main(void)
09 {
10     printf("%d\n", x + func());
11     return 0;
12 }
```

이와 같은 경우 C 컴파일러에서는
피연산자의 우선순위가 정해지지 않아
컴파일러 마다 다른 결과를 낼 수도 있음
(따라서 소스 1은 문제가 있음)

✓ 10행의 `x + func()`에서 왼쪽 피연산자인 `x`가 먼저 평가

x	+	func()
10		30

✓ 오른쪽 피연산자인 `func()` 함수가 먼저 평가

x	+	func()
20		30

Operand Evaluation Order (in JAVA)

```
01 public class operand {
02     static int x=10;
03     public static int func() {
04         x = 20;
05         return 30;
06     }
07     public static void main(String[] args) {
08         System.out.println(x + func());
09     }
10 }
```

[소스 2]

JAVA에서는 피연산자의 평가 순서를
왼쪽에서 오른쪽으로 정해 놓고 있으므로
소스 2는 문제가 없음

Functional side effects

피연산자의 평가 순서에 따라 다른 결과가 나오는 문제를 해결하기 위한 방법으로,
피연산자의 평가 순서(left-to-right or right-to-left order)를 정해놓음으로써
함수의 부작용을 방지할 수 있음

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

프로그램 액션에 영향을 미치지 않는다

```
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**

fun 함수가 b 또는 c의 값을 변경하는 side effects를 갖게 된다면,
참조의 투명성이 위반된다.

→ 참조의 투명성 개념은 함수의 부작용(functional side-effects)과 연관됨

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
(more details in ch15)

Overloaded Operators

- Use of an operator for more than one purpose is called **operator overloading**
- Some are common (e.g., **+** for `int` and `float`)
 - In Java, **+** is used for string catenation
- **'&'** operator (C++)
 - `a && b`, `a &b`; //binary operator: logical operator, Bit operator
 - ① & - 대응 되는 모든 비트(bit)의 조건을 모두 검사(모든 bit를 알아야 연산, bool 타입, 숫자 타입에 사용)
 - ② && - 첫 번째 조건이 거짓이면 두 번째 조건은 검사 하지 않음(bool 타입의 경우만)
 - `x = &y`; //unary operator: address-of operator
- Some are potential trouble (e.g., ***** in C and C++)
 - Loss of compiler error detection (omission of an operand should be a detectable error)
 - `a&b` → `&b`: 컴파일러는 주소연산자로 인식
 - 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) (p. 349)
 - In C++ (중복이 허용되지 않은 연산자)
 - class or structure member operator(.)
 - Scope resolution operator(::)
 - ex) ::a (global variable a in local area)
 - More details in ch.9
 - Potential problems:
 - Users can define nonsense operations
 - Readability may suffer, even when the operators make sense

Type Conversions

- A **narrowing conversion** (축소 변환) 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 (not safe)
- A **widening conversion** (확장 변환) 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 (safe, but some precision may be lost)
 - fig 6.1 (p.268), single precision (32 bits)

부동소수점 참조(<http://soen.kr/lecture/ccpp/cpp2/18-1-4.htm>)

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

```
Int a;  
float b, c, d;  
...  
d = b * a;  
// because of a keying error ( c → a)  
// the value of int operand(a) is coerced to float  
// int → float (widening conversion)
```

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

Overflow vs Underflow

- Overflow

- **Buffer overflow** : 할당된 버퍼를 넘어서 쓰기가 발생하는 경우 (stack overflow, heap overflow)
- **Arithmetic overflow** : 연산의 결과값이 저장할 레지스터 (register)나 저장 위치보다 큰 경우 발생

- Underflow

- **Buffer underflow** : 빈 버퍼로부터 읽기가 발생하는 경우
- **Arithmetic underflow** : 산술연산의 결과가 취급할 수 있는 수의 범위보다 작아지는 상태

Overflow vs Underflow (continued)

```
#include <stdio.h>
void main() {
    int i = 32767;
    printf("%d %d \n", i, i+1); /* overflow */

    i = -32768;
    printf("%d %d \n", i, i-1); /* underflow */
}
```

/* 최대 수인 int형에 1을 더할 경우 32768을 표시할 방법이 없으므로,
가장 작은 수인 -32768부터 다시 숫자를 표시하는 overflow 발생 */

/* 가장 작은 수인 -32768에 1을 빼는 경우 int형 범위를 벗어나므로,
가장 큰 수인 32767로부터 다시 숫자를 표시하는 underflow 발생*/

/* 문자형, 실수 형에서도 overflow, underflow 현상은 발생함 */

Ref) <http://v0nsch3lling.tistory.com/55>

Overflow vs Underflow (continued)

References

<http://en.wikipedia.org/wiki/Overflow>

http://en.wikipedia.org/wiki/Buffer_overflow

http://en.wikipedia.org/wiki/Stack_overflow

http://en.wikipedia.org/wiki/Heap_overflow

http://en.wikipedia.org/wiki/Arithmetic_overflow

<http://en.wikipedia.org/wiki/Underflow>

http://en.wikipedia.org/wiki/Buffer_underflow

http://en.wikipedia.org/wiki/Arithmetic_underflow

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

- C-based(!=)

- Ada(/=)

- Lua(~=)

- Fortran 95+ (.NE or <>)

- ML, F#(<>)

“7” == 7
이 경우 강제로 타입변환 해 줌

“7” === 7
이 경우 타입변환 안해줌 >같은 타입을 써야만 함

- JavaScript and PHP have two additional relational operator, **===** and **!==** (p. 353)

- Similar to their cousins, **==** and **!=**, except that **they do not coerce their operands**
- Ruby uses **==** for equality relation operator that **uses coercions** and **eq?** for those that **do not**

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: **(P.354)**
 - 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
(Suppose Java did not use short-circuit evaluation)

인덱스가 커지다가 같아지는 순간 단축평가를 하면 앞만 평가하고 넘어가지만
단축평가를 하지 않을 경우 뒤도 평가해야하는데 평가 할 경우 배열 인덱스범위를 넘어감 > 에러발생

```
index = 0;
while ((index < listlen) && (list[index] != key))
    index++;
```

```
// When index = listlen, list[index] will cause an indexing problem
// (assuming that list, which has listlen elements, is the array to be searched
// and key is the searched-for value)
```

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)**

a <= b 경우에만 두 번째 산술 식에 표현된 b값이 변경

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 languages
 - 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

로직상 차이가 없다면 prefix를 사용하는게 postfix를 사용하는 것 보다 더 좋다

- 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
              // is equivalent to -(count++)
```

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 (e.g. - p. 359)

산술 식을 피 연산자(operand)로 사용했을 경우?
`a = b +(c=d/b) -1;`

Assign d/b to c
Assign b + c to temp
Assign tem - 1 to a

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 (skip~)

- 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**
- **강제변환**
In Java and C#, only **widening assignment coercions** are done
: int → float (O), float → int (X)
- In Ada, there is no assignment coercion

Summary

- Expressions 연산자 우선순위에 따라서 > side effect가 발생할 수 있다(절댓값이 큰 음수 양수를 더할때)
- Operator precedence and associativity
- Operator overloading 가능한 안쓰는게 좋다
- Mixed-type expressions
- Various forms of assignment