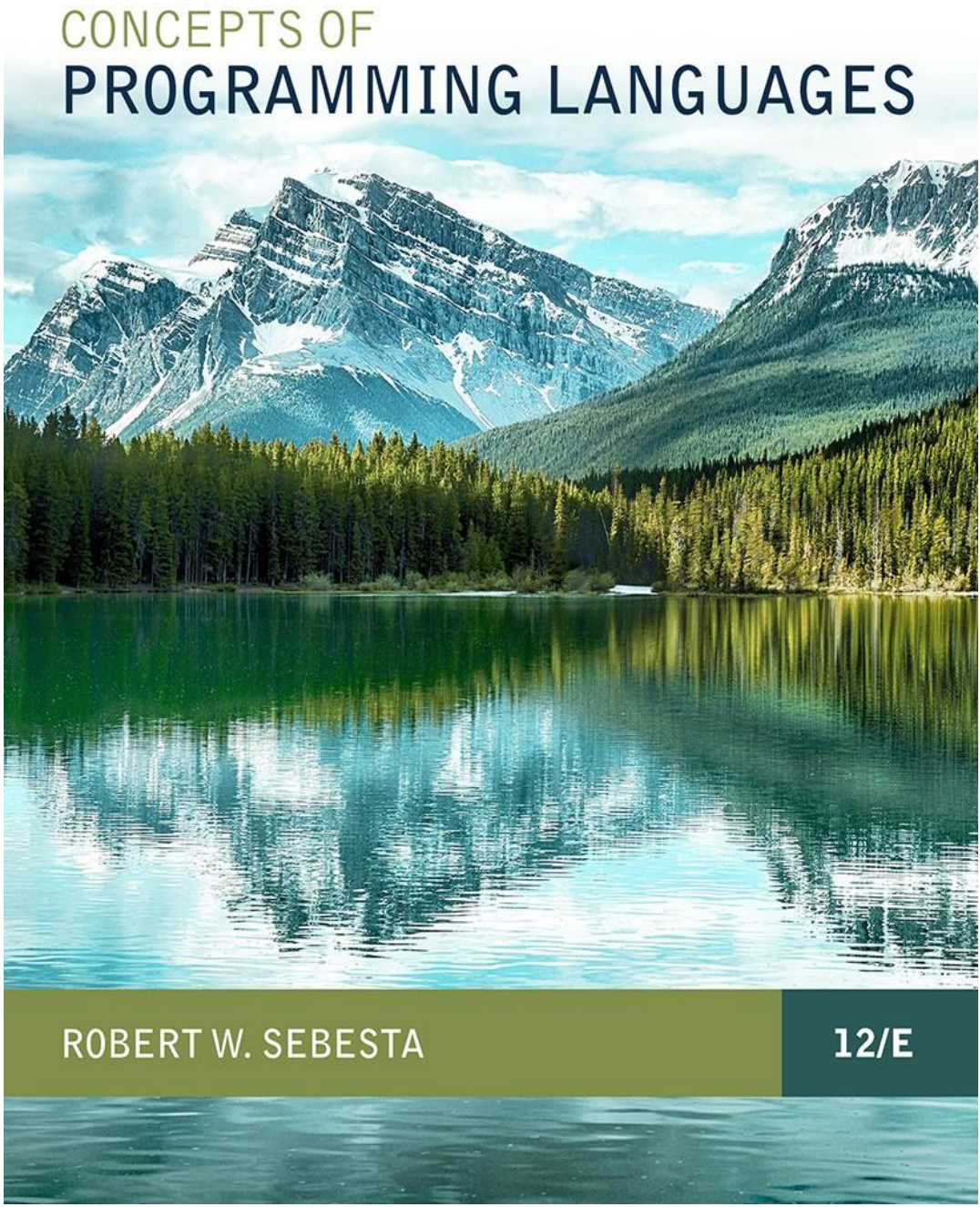
Chapter 7

Expressions and Assignment Statements



ISBN 0-321-49362-1

Chapter 7 Topics

- 서론
- 산술식
- 중복 연산자
- 타입 변환
- 관계식과 불리안식
- 단락회로 평가
- 배정문
- 혼합형 배정문

서론

- 식(expression)은 프로그래밍 언어에서 계산을 명세하는 기본적인 수단
- 식의 평가는 연산자와 피연산자의 평가 순서를 잘 알아야 함
- 명령형 언어의 핵심은 배정문의 역할

산술식

- 산술식 평가는 초기 프로그래밍 언어의 개발 동기였음
- 산술식은 연산자, 피연산자, 괄호, 함수 호출로 구성

- 산술식의 설계고려사항
  - 연산자 우선순위규칙
  - 연산자 결합규칙
  - 피연산자 평가순서
  - 피연산자 평가 부작용
  - 연산자 중복 허용
  - 표현식에서 타입 혼합

산술식 : 연산자 평가순서

1) **연산자 우선순위 규칙 (*operator precedence rules)*** :
다른 우선 순위를 갖는 연산자들이 평가되는 순서

2) **연산자 결합 규칙(*operator associativity rule)*** :
동일한 수준의 우선순위를 가지면서 인접된 두 개의 연산자를 포함할 때의 순위

※ 우선순위와 결합 규칙은 ( )로 override

Ruby 와 Scheme의 식

- Ruby(순수객체지향언어)
  - 모든 산술, 관계, 배정 연산자, 배열 인덱싱, 쉬프트, 비트단위 논리 연산자 등이 매소드로 구현

    (ex) a + b
  - 연산자는 응용 프로그램에서 override 가능 (연산자 재정의)
- Scheme (Common LISP)
  - 모든 산술과 논리 연산은 부프로그램 호출로 수행

    (ex) a + b * c 은 (+ a (* b c))으로 코딩,

    여기서 +와 *는 함수이름이다.

산술식 : 피연산자 평가 순서

- *피연산자 평가 순서*
  1. 변수: 메모리로부터 값을 가져옴
  2. 상수: 메모리로부터 가져오거나 기계어 명령어에 있음
  3. 괄호 식: 괄호 안의 모든 피연산자와 연산자들을 먼저 평가
  4. 가장 흥미로운 경우는 피연산자가 함수호출문인 경우임

산술식: 잠재적인 부작용

- 함수적 부작용*(Functional side effects)* : 함수가 양방향 파라미터나 비지역변수를 변경할 때 피연산자의 평가 순서에 따라 다른 결과가 발생하는 문제

1) 식에서 참조된 함수가 식의 또 다른 피연산자를 변경시킬 때

```
a = 10;
/* fun()은 매개변수의 값을 20으로 변경하고
   10을 반환한다고 가정    */
b = a + fun(&a);
```

여기서 a가 먼저 평가된다면 b의 값은? 20

만약 fun(a)가 먼저 평가된다면 b의 값은? 30

산술식: 잠재적인 부작용(continued)

2) 함수가 식에 포함된 비지역변수(전역변수)를 변경할 때

```
int a=5;  // 전역변수
int fun1()
{ a=17;
  return 3; }
void main()
{ a = a + fun1();
}
```

여기서 a가 먼저 평가되면 결과는? 8

만약 fun1()이 먼저 평가되면 결과는? 20

함수적 부작용

- 두 가지 해결방법

  1) 함수적 부작용을 허용하지 않도록 언어 정의를 한다.
     - 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) 피연산자 평가 순서를 특정 순서로 고정되도록 언어정의를 한다.
     - **Disadvantage** : 컴파일러 최적화를 제한함
     - Java는 피연산자들이 left-to-right 순서로 평가되도록 고정

참조 투명성

- 프로그램에서 동일한 값을 갖는 임의의 두 개 식이 프로그램의 행동에 영향을 미치지 않으면서 프로그램의 임의의 위치에서 서로 다른 식으로 대체될 수 있다면 그 프로그램은 참조투명성(*referential transparency*)의 특성을 가짐

```
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

참조 투명성 (continued)

- 참조 투명성의 장점
  - 참조 투명성을 가지면 프로그램의 의미를 이해하기 더 용이

- 순수 함수형 언어의 프로그램은 변수들을 갖지 않기 때문에 참조 투명하다.
  - 함수의 상태(state)는 지역 변수에 저장되는데, 함수형 언어의 함수는 상태를 가질 수 없다.
  - 만약 함수가 외부 값을 사용하면 그 값은 상수이어야 한다.

    따라서 함수의 값은 오직 매개변수의 값에만 종속됨.

중복정의된 연산자

- 한가지 이상의 목적으로 연산자를 사용하는 것이 연산자 중복(*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

중복 연산자 (continued)

- C++, C#, F# 는 사용자 중복 연산자 허용
  - 분별있게 사용될 때 판독성 향상 가능

    (매소드 호출 불필요, 식의 자연스러운 표현)

    (e.g.) 스칼라 정수와 정수 배열 간의 *, + 정의되었다면

    MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))

    → A * B + C * D
  - Potential problems:
    - Users can define nonsense operations
    - Readability may suffer, even when the operators make sense

타입 변환(Type Conversions)

- 축소 변환(*narrowing conversion*) : 값을 그 원래 타입의 모든 값들을 포함할 수 없는 타입으로 변환

  (ex) **`float`** to **`int`**

- 확장 변환(*widening conversion*) : 값을 적어도 그 원래 타입의 모든 값들의 근사치로 포함할 수 있는 타입으로 변환; 거의 안전하나 정확성 감소 가능

  (ex) **`int`** to **`float`**: **십진수 9자리 -> 소수점 7자리 정확성**

타입 변환 : 혼합형

- 혼합형 식(*mixed-mode expression*) : 타입이 다른 피연산자로 구성된 식
- 강제변환(*coercion*)은 컴파일러의 묵시적 타입 변환임
- 강제변환의 단점 :
  - 컴파일러의 타입 오류 탐지 능력을 감소시킴
- 대부분 언어의 경우, 식에서 모든 수치 타입들은 확장 변환을 이용하여 강제 변환됨
- ML과 F# 은 식에서 강제변환 없음

명시적 타입 변환

- C-기반 언어에서 캐스팅(*casting*)
- Examples
  - `C: (int)angle`
  - `F#: float(sum)`

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

식의 오류

- 원인
  - 산술연산의 본질적인 한계

    (ex) division by zero
  - 컴퓨터 산술 연산의 한계

    (ex) overflow
- 이런 오류는 실행-시간 오류이며, 때로는 예외라고 함
- Often ignored by the run-time system

관계식과 불리안 식

- 관계식
  - 관계 연산자와 여러 타입의 피연산자 사용
  - 관계식의 평가 결과는 불리안 타입
  - 연산자 기호는 언어들 마다 다소 상이
    (ex. 비동등 연산자 : `!=`, `/=`, `~=`, `.NE.`, `<>`)

- JavaScript와 PHP는 관계 연산자 추가 제공(`===` , `!==`)
  - `==`, `!=`과 유사하나, 피연산자의 타입을 강제변환하지 않음
    (ex. "7" == 7 vs "7" === 7)
  - Ruby는 강제변환을 갖는 동등연산으로 `==` 을 사용하며,
    강제변환이 없는 동등은 `eql?`을 사용함

관계식과 불리안 식 (continued)

- 불리안 식
  - 피연산자가 불리안 타입이고 결과도 불리안 타입
  - Example operators
- C89는 no Boolean type

  → **`int`** 형의 0은 false

  0이 아닌 수는 true
- C의 식에서 특이한 점은 **`a < b < c`** 이 적법, 결과는 기대한 것과 다름 :
  - 왼쪽의 관계연산자가 평가되어 0 or 1이 나옴
  - 이 결과값이 변수 c와 비교됨(b와 c간의 비교는 없음)

단락 회로 평가(Short Circuit Evaluation)

- 식의 모든 연산자들을 평가하지 않고 그 결과를 결정하는 평가

  (Ex) `(13 * a) * (b / 13 - 1)`

  ⇒ a가 0이면, `(b/13 - 1)`연산 불필요

  (Ex) `(a >= 0) && (b < 10)`

  ⇒ a<0이면 나머지 관계식 연산 불필요

- 단락회로평가가 아닌 경우의 문제(Java의 예)

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

⇒ index=length일 때, LIST[index] 은 indexing problem 야기 (LIST의 길이가 length - 1 이라면)

단락 회로 평가(continued)

- **C, C++, Java** : 불리안 연산자(`&&`, `||`)에 대해서는 단락 회로 평가를 사용하나 비트단위의 불리안 연산자(`&`, `|`)에 대해서는 제공하지 않음
- **Ruby, Perl, ML, F#, Python** : 모든 논리 연산자는 단락 회로 평가로 수행됨
- 단락 회로 평가는 식의 부작용 문제를 노출
  (ex) `(a > b) || (b++ / 3)`에서 `b`의 값

배정문

- 일반 구문

  <target_var> <assign_operator> <expression>

- 배정 연산자

  = Fortran, BASIC, the C-based languages

  := Ada

- = 은 동등 관계 연산자로 중복 정의되는 것을 조심!!
  (C-기반 언어에서 동등 연산자로 ==을 사용함)

배정문: 조건 목적지

- 조건 목적지 (Perl에서)

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

다음과 동일함

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

식으로서의 배정문

- C-기반 언어, Perl, JavaScript에서 배정문은 결과를 생성하며, 다른 식의 피연산자로서 사용될 수 있음

  (ex) **while** ((ch = getchar())!= EOF){...}

  ⇒ ch = getchar() is carried out; the result(assigned to ch) is used as a conditional value for the **while** statement

  (ex) a= b + (c = d / b) - 1; // 다른 유형의 식 부작용 초래

- If(x==y) … 와 if(x=y) … :

  ⇒ C와 C++은 컴파일러 오류 탐지 불가

  그러나 Java와 C#은 조건식에 불리안식만 허용

다중 배정문

- 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);
```

함수형 언어의 배정문

- 함수형 언어에서 식별자는 단지 값들의 이름임
- ML
  - 이름은 **val 선언문**으로 값에 바인딩된다.

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

  - 만약 또 다른 `val fruit` 문이 온다면 그것은 새로운 다른 이름이다.

혼합형 배정문

- 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