7장 식과 배정문

2020. 6. 11

순천향대학교 컴퓨터공학과

목차

산술식

- 산술식은 연산자, 피연산자, 괄호, 함수 호출 등으로 구성
- 산술식의 설계 고려사항
  - 연산자 우선순위 규칙은?
  - 연산자 결합 규칙은?
  - 피연산자 평가 순서는?
  - 피연산자 평가 부작용(side effect)에 대해서 제한이 있는가?
  - 사용자-정의 연산자 중복을 허용하는가?
  - 식에서 혼합형이 허용되는가?

연산자 평가순서

- 언어는 연산자 평가 순서를 명세한다.
  - 연산자 우선순위
  - 결합규칙
  - 괄호

피연산자 평가 순서

- 피연산자 유형
  - 변수
  - 상수
  - 괄호에 포함된 식
  - 함수 호출

- 연산자의 모든 피연산자가 부작용(side effect)을 갖지 않으면, 피연산자의 평가 순서는 무관하다.

함수의 부작용

- 함수의 부작용(functional side effects)은 함수가 양방향 매개변수나 전역변수 변경시 발생

- 함수의 부작용 문제
  - 식에서 참조된 함수가 식의 다른 피연산자를 변경할 때 발생
  - 피연산자 평가 순서에 영향을 미침

```
a = 10;
b = a + fun(a); // fun()이 a를 변경하면?
```

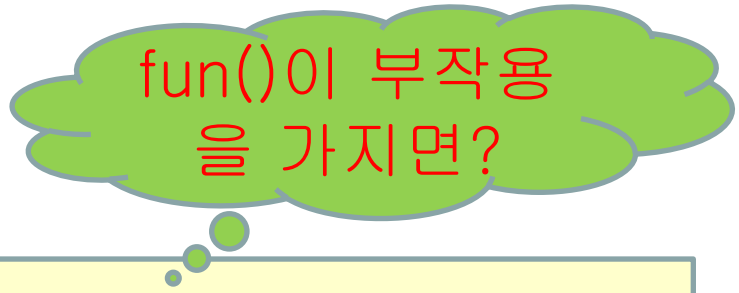
함수의 부작용 해결책

- 함수의 부작용을 허용하지 않는 것
  - 양방향 매개변수, 비 지역변수 참조를 불허
  - 유연성 저하

- 피연산자 평가 순서를 정하는 것
  - Java : 왼쪽부터 오른쪽
  - 컴파일러 최적화 제한

참조 투명성

- 참조 투명성(referential transparency): 프로그램에서 동일한 값을 갖는 임의 2개 식이 프로그램의 행동에 영향을 미치지 않으면서 프로그램의 임의 위치에서 서로 대체 가능하면, 프로그램은 참조 투명성의 특징을 갖는다고 말한다.




```
result1 = (fun(a) + b) / (fun(a) – c);
temp = fun(a);
result2  = (temp + b) / (temp – c); // result1 == result2?
```

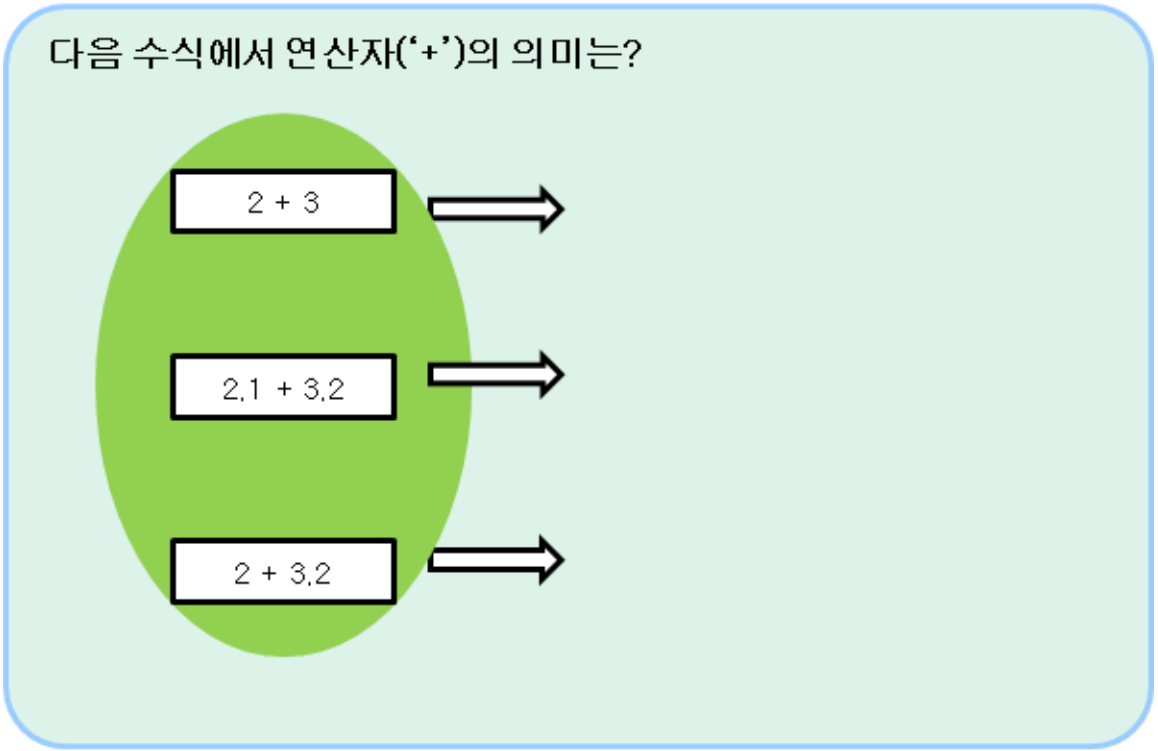
참조 투명성(2)

- 참조 투명한 프로그램의 장점
  - 프로그램 의미를 이해하는 것이 더 쉬움

- 순수 함수형 언어로 작성된 프로그램은 참조 투명하다
  - 변수가 없다
  - 함수의 값은 단지 매개변수에만 종속

중복 연산자

- 연산자가 한 가지 이상의 목적으로 사용되면, 이를 연산자 중복(operator overloading)이라 한다.
  - Ex. C의 '+'



중복 연산자 (2)

- 어떤 것은 일반적
  - C의 '+'

- 어떤 것은 문제초래 가능성
  - C의 '&', -: 이항 vs 단항 연산자
  - 중복된 의미간의 관련성에 따른 판독성 영향

사용자-정의 중복 연산자

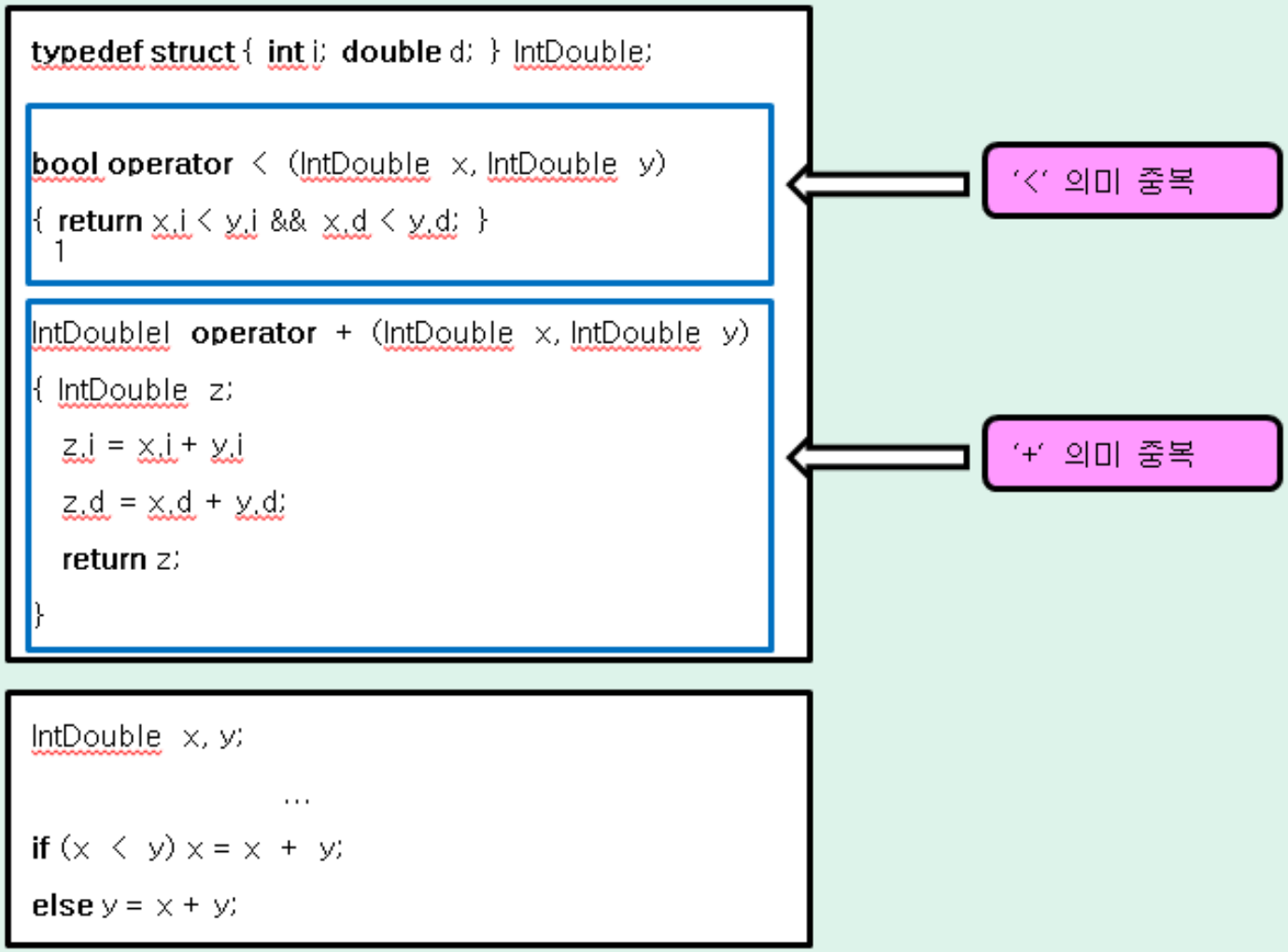
- 사용자가 중복 연산자를 정의하는 것을 허용
- 언어 예: C++, C#
  - In C++

다음 프로그램은 올바른가?

```
typedef struct { int i; double d; } IntDouble;
IntDouble  x, y;
                    ...
if (x  <  y)  x = x  +  y;
else  y = x + y;
```

예 제

C++는 사용자-정의 연산자 중복을 위해서 operator 키워드를 제공한다.



평가: 사용자-정의 중복 연산자

- 분별있게 사용되면 판독성 향상

  A * B + C * D // A, B, C, D는 행렬

  vs Madd(Mmult(A,B), Mmult(C,D))

- 판독성 저하 요인
  - 분별없이 연산자를 중복 정의하면?

    ex. '+'를 곱셈 의미로 중복

  - 중복된 연산자의 의미를 파악하는 것이 필요

- 소프트웨어가 여러 모듈로 구성될 때, 다른 모듈이 동일한 연산자를 다른 의미로 중복 정의하면?

타입 변환

- 축소 변환(narrowing conversion)은 원래 타입에 속한 모든 값들의 근사치마저도 저장할 수 없는 타입으로 변환
  - 예: float => int
  - 안전하지 않음

- 확장 변환(widening conversion)은 적어도 원래 타입의 모든 값들의 근사치를 포함할 수 있는 타입으로 변환
  - 예: int => float
  - 거의 안전하나 정확성 감소 초래

    ex) int (32비트) => float (32비트: 8(exponent)+23(fraction))

    9자리 십진수 => 7자리 십진수

혼합형 식

- 혼합형 식(mixed-mode expression)은 한 연산자가 다른 타입의 피연산자들을 갖는 식
- 이를 허용하는 언어는 묵시적 피연산자 타입 변환을 허용 (확장 변환으로)
- 묵시적 타입 변환은 컴파일러에 의해서 수행되는 타입 강제변환

명시적 타입 변환

- 명시적 타입 변환은 프로그램어에 의해서 명시적으로 요구하는 타입 변환
  - 대부분의 언어는 축소 변환, 확장 변환을 위한 명시적 변환 제공
  - C-기반 언어에서 캐스트(cast)라 부른다.
  - 예:

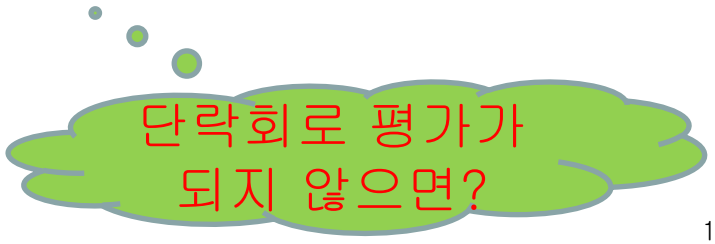
```
(int) angle    // in C

Float (sum)  // in Ada
```

단락회로 평가

- 단락회로 평가(short-circuit evaluation)는 식에 포함된 모든 연산자나 피연산자가 평가되지 않고서 식의 값이 결정되는 평가

```
(13 * a) * (b /  13 -1)      // what if a = 0?
```

```
index = 1;
while (index <= length) && (LIST[index] != value)
      index++;
```



단락회로 평가 (2)

- 식에 부작용이 존재하면?

```
(a > b) || (b++ / 3) // 식이 평가될 때마다
                     // b의 값이 변경된다고 가정하면?
```

- 언어 예
  - C, C++, Java: 부울 연산자 &&, ||에 대해서 단락회로 평가 적용
  - Perl, Python에서도 적용

배정문

- 배정문이 혼합형(mixed-mode)일 수 있다.

```
int a, b;
float c;

C = a / b;
```

- 배정시 타입 강제변환
  - 임의 수치 값이 임의 수치 타입 변수에 배정가능한가?
  - 단지 확장 타입 강제 변환만 허용하는가?
  - 배정 타입 강제변환을 허용하지 않는가?