

프로그래밍 언어론

1. 함수형언어 개요 (Introduction to Functional Languages)

컴퓨터공학과
이만호



함수형언어 개요

학습 목표

수학에 기반을 두고 있는 함수형언어의
일반적 특성에 대해서 학습한다.

학습 내용

- 수학적 함수
- Functional Form
- 함수형언어의 특성
- Tail Recursion
- 함수형언어의 예



목 차

- 들어가기
- 학습하기
 - 수학적 함수
 - Functional Form
 - 함수형언어의 특성
 - Tail Recursion
 - 함수형언어의 예
- 평가하기
- 정리하기



알고가기



수학에서 어떤 함수의 domain(정의역)과 range(치역)에 관한 설명으로 맞지 않는 것은?

- a. 함수는 domain의 원소를 range의 원소로 mapping하는 것이다.
- b. 함수의 domain과 range는 항상 다르다.
- c. 함수의 domain의 원소들이 함수일 수 있다.
- d. 함수의 range의 원소들이 함수일 수 있다.

확인



| 명령형언어 vs. 함수형언어

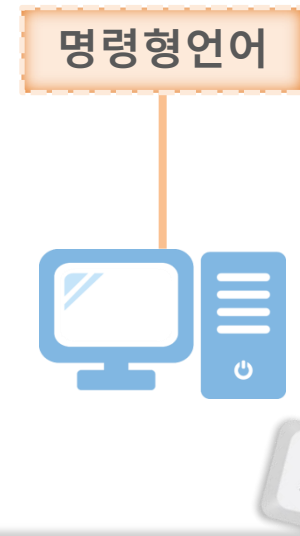
명령형언어(Imperative Language)

- ➔ von Neumann 구조에 기반을 두고 있음
- ➔ 프로그램 실행의 효율성이 가장 중요하게 생각됨
- ➔ Software를 자연스럽게 개발하기에는 제약이 많음
 - ▶ 언어 설계 개념이 문제나 사람 중심이 아니라, hardware 중심
- ➔ Syntax와 semantics가 복잡함



함수형언어(Functional Language)

- ➔ 수학적 함수에 기반을 두고 있음
 - ▶ 철저히 이론에 근거하므로, 사용자가 쉽게 사용할 수 있다.
- ➔ 프로그래밍 실행될 컴퓨터 구조(architecture)와는 상관이 없다.
 - ▶ 프로그램 실행이 효율적이지 않다.
- ➔ Syntax와 semantics가 간단함



| 수학적 함수

함수(function)

Mapping : $D \rightarrow R$

D : Domain set (정의역),

R : Range set (치역)

수학적 함수

- 함수의 한 표현 방법으로 함수 이름이 있다.
- 함수 정의: `square(x) = x * x`
- 함수값 계산(application): `square(3) = 3 * 3`

Lambda Expression

- 함수의 한 표현 방법으로 함수 이름이 없다.
- 함수 정의: `λ(x) x * x`
- 함수값 계산(application) : `(λ(x) x * x)(3) = 3 * 3`



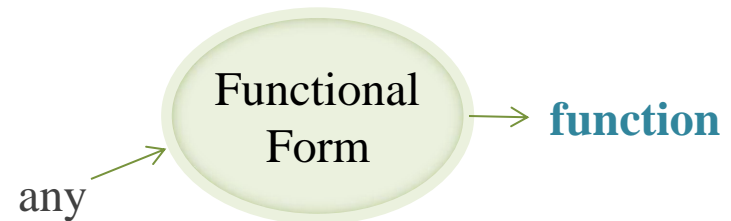
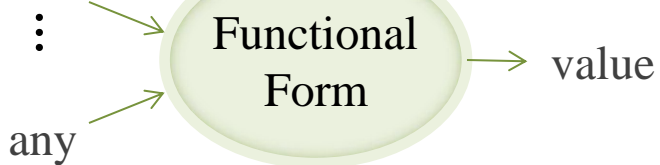
| Functional Form

Higher-order function 이라고도 한다.

조건 다음 중 한 가지 이상을 만족해야 한다.

- Parameter로 1 개 이상의 function을 받는다.
- Function의 결과가 function이다

function



예

Functional Form

- ▶ Function Composition
- ▶ Apply-to-all



| Function Composition

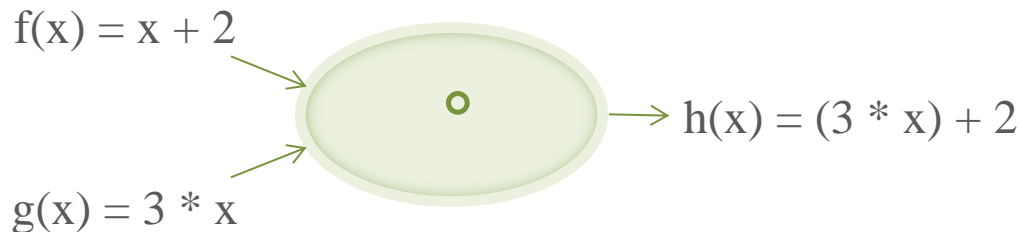
Function Composition(합성함수)

- ➔ Operator: \circ
- ➔ Parameter
 - ▶ 2개의 function (f와 g라고 하자)
- ➔ 결과
 - ▶ 둘째 function의 값을 먼저 구하고, 그 결과를 첫째 function에 적용하여 구한다.

예

$$h = f \circ g \quad \text{where } f(x) = x + 2, \quad g(x) = 3 * x$$

$$h(x) = f \circ g(x) = f(g(x)) = (3 * x) + 2$$

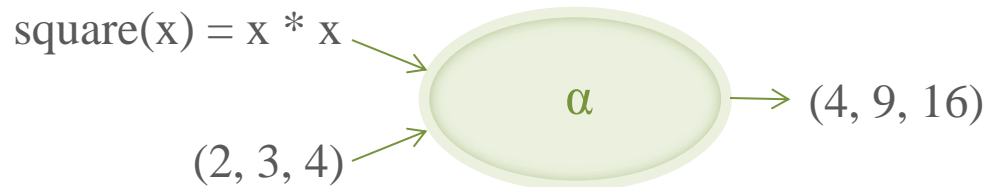


| Apply-to-all

Apply-to-all

- ➔ Form: α
- ➔ Parameter
 - ▶ 1개의 function('square'라고 하자)
 - ▶ 1개의 list
- ➔ 결과
 - ▶ list의 각 element에 'square'를 적용해서 얻은 결과의 list

예

 $\text{square}(x) = x * x$
 $\alpha(\text{square}, (2,3,4)) = (4, 9, 16)$


| 함수형언어의 특성



함수형언어의 목적

- ➔ 수학적 함수와 최대한 비슷하게 흉내 내기



프로그램의 형태

- ➔ 함수의 정의(definition)와 함수의 적용(application)으로 구성됨
 - ▶ 한 함수의 결과는 다른 함수의 parameter로 사용됨
 - ▶ Assignment 문장을 사용하지 않음



함수의 결과로 return되는 값의 type에 제한이 없음



변수를 전혀 사용하지 않음

- ➔ 반복실행(repetition, loop)이 불가능
 - ▶ 반복실행은 반복실행을 제어하는 변수를 기반으로 하고 있음
 - ▶ Recursion을 사용해서 반복실행 효과를 얻음 → 실행시간이 오래 걸림
- ➔ Side-effect(부수효과)가 발생하지 않음
 - ▶ Referential Transparency(참조투명성)가 보장됨



대부분의 함수형언어는 interpreter로 구현됨



주로 다루는 자료구조는 list임



Tail Recursion을 지원함 : 다음에 자세히 설명함



| Tail Recursion

Tail Recursion의 특성

- ➔ Function이 return하는 값이 단순 값이거나 단순 recursive call인 형태
- ➔ Compiler가 iteration으로 쉽게 변환할 수 있는 recursive function
 - ▶ 함수형언어에만 해당됨

Euclidean 호제법

```
int gcd(int m, int n)
{
    if (n==0) return m;
    if (m>=n) return gcd(n, m%n);
    else      return gcd(n, m);
}
```

```
gcd(5083, 4807)
= gcd(4807, 276)
= gcd(276, 115)
= gcd(115, 46)
= gcd(46, 23)
= gcd(23, 0)
= 23
```

1	5083	4807	17
	4807	4692	
2	276	115	2
	230	92	
2	46	23	
	46		
	0		



| Non-Tail Recursion

Non-Tail Recursion의 특성

- ➔ Recursive call의 결과로 받은 값에 추가적인 연산을 수행하고 그 결과를 return한다.
- ➔ Compiler가 iteration으로 쉽게 변환할 수 없는 recursive function
- ➔ Compiler는 non-tail recursion 을 자동적으로 tail recursion으로 변환해 줄 수도 있다.
(함수형언어에만 해당됨)

```
int fact(int n)
{
    if (n==0) return 1;
    return n * fact(n-1);
}

fact(3)
= 3 * fact(2)
      = 2 * fact(1)
            = 1 * fact(0)
                  = 1 * 1 = 1
      = 2 * 1 = 2
= 3 * 2 = 6
```

```
int factR(int n, int r)
{
    if (n==0) return r;
    return factR(n-1, r*n);
}

fact(n) = factR(n,1)

fact(3) = factR(3,1)
= factR(2,1*3) = factR(2,3)
= factR(1,3*2) = factR(1,6)
= factR(0,6*1) = factR(0,6)
= 6
```

| 함수형언어의 예 - 1

Lisp

- ➔ 최초의 함수형언어
 - ▶ 최근 발전된 함수형언어의 개념을 반영하지 못함
- ➔ 명령형언어 형태의 프로그램을 작성할 수 있는 기능 지원
 - ▶ 변수, assignment, 반복실행

Common Lisp

- ➔ Lisp 계열의 모든 함수형언어가 지원하는 기능 대부분을 지원함
 - ▶ 매우 방대하고 복잡한 언어가 됨 — Scheme과 반대
- ➔ 함수형언어 중에서 가장 널리 사용됨 (특히 인공지능 분야)

Scheme : 다음 강좌에서 자세히 다룰 것임



| 함수형언어의 예 - 2

ML

- ➔ Static scope rule만 지원
- ➔ Strongly typed 언어임.
 - ▶ Type coercion(강제 변환) 하지 않음
- ➔ Type이 지정되지 않은 변수에 type inferencing(type 추론) 적용
- ➔ Syntax가 명령형언어와 비슷함
 - ▶ infix 형태의 수식, if-then-else 구조

Haskell

: ML과 비슷함

- ➔ 함수가 polymorphic function(다형함수)일 수 있음

```
sum [] = 0
sum (h:T) = h + sum T    // h:head of a list, T:tail of a list
```

- ➔ Lazy evaluation (Eager evaluation이 아님)
 - ▶ 실제로 사용되지 않는 수식은 절대로 계산되지 않음
- ➔ Call-by-name을 구현할 수 있음
- ➔ 무한수열을 생성할 수 있음

```
positive = [0..]
evens = [2, 4..]
squares = [n * n | n <- [0..]] ; [0, 1, 4, 9, ...]
```



평가하기

마지막으로 내가 얼마나 이해했는지를 한번 확인해 볼까요?
총 2문제가 있습니다.

START



평가하기 1

1. 함수형언어와 명령형언어를 비교한 것으로 가장 적당하지 않는 것은?

- a. 함수형언어가 컴퓨터 hardware를 더 고려한다
- b. 함수형언어가 수학의 특성을 더 잘 반영한다.
- c. 함수형언어로 작성된 프로그램의 실행 속도가 더 느리다.
- d. 함수형언어로 프로그램을 작성하기가 더 용이하다.

확인



평가하기 2

2. 함수형언어의 특성으로 적당하지 않은 것은?

- a. 변수를 전혀 사용하지 않고 프로그램을 작성할 수 있다.
- b. 함수의 결과로 return되는 값의 type에 제한이 없다.
- c. 반복 실행이 프로그램 작성에 매우 많이 사용된다.
- d. 순수한 함수형언어는 side-effect를 유발하지 않는다.

확인



정리하기

➡ 함수

- ▶ 수학적 함수
- ▶ Lambda Expression

➡ Functional Form

- ▶ Function Composition
- ▶ Apply-to-all

➡ 함수형언어의 특성

- ▶ 수학적 함수와 최대한 비슷

➡ Tail Recursion

- ▶ Function이 return하는 값이 단순 값이거나 단순 recursive call인 형태

➡ 함수형언어의 예

- ▶ Lisp, Common Lisp, Scheme, ML, Haskell



“ 강의를 마치겠습니다. 수고하셨습니다. ”

