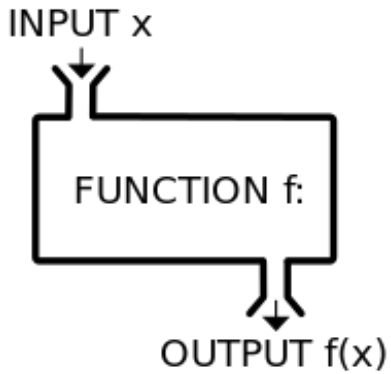
13. 함수형 언어

숙명여대 창병모

13.1 함수형 언어

함수형 언어의 시작

- 프로그램
  - 단순히 입력으로부터 출력을 계산하는 '블랙박스'라고 볼 수 있음.
  - 이러한 관점에서 프로그램은 수학적 함수로 표현할 수 있다.



- 함수형 언어
  - 수학적 함수를 기반으로 하는 언어
  - 프로그램이 하는 일을 수학적 함수의 계산으로 취급한다.
  - 매개변수로 입력을 받아 처리한 후에 반환값을 출력하는 함수

함수형 언어(Functional language)

- 중요 개념
  - 함수 정의(function definition)
  - 함수 호출 = 함수를 값에 적용(application)
  - 매개변수 전달(parameter passing)
  - 값 반환(return value)

- 특징
  - 변수 및 대입문이 없음
  - 루프 같은 반복문은 없음
  - 자기호출(recursion)에 의한 반복
  - 함수는 일등급 값으로 다른 값처럼 인자로 사용될 수 있고 반환 값으로도 사용될 수 있다.

함수형 언어(Functional language)

- 예제 1(S)

함수 정의

```
fun int f(int n)  return n*n;
```

함수 호출

```
f(5);
```

- 예제 2(ML)

함수 정의

```
fun fact(n : int) : int =
        if n <=1 then 1
        else n * fact(n-1);
```

함수 호출

```
fact(5);
```

- 장점
  - 기계 모델과 무관
  - 프로그램을 함수로 보는 일관된 관점을 제공한다.
  - 수학을 기반으로 프로그램의 의미를 명확하게 정의할 수 있다.
- 예
  - Scheme, ML, Haskell

13.2 람다 계산법

람다 계산법

- 람다 계산법(Lambda calculus)
  - 함수형 언어의 기본 이론

- 람다식(Lambda expression)
  - 익명 함수(anonymous function)를 표현하기 위한 식

f(x) = … → 익명이아님

```
E → c          # 상수
  | x          # 변수로 x, y, x1, x2 등으로 사용
  | λx.E       # 익명 함수 정의(x가 변수이고 E가 람다식일 때)
  | E1 E2      # 함수 적용(E1은 함수, E2는 인수를 나타내는 람다식)
```

(보통) 자

- 람다식은 일등급 값(first class value)
  - 매개변수로 전달할 수 있다.
  - 반환값으로 사용할 수 있다.

수식과 함수

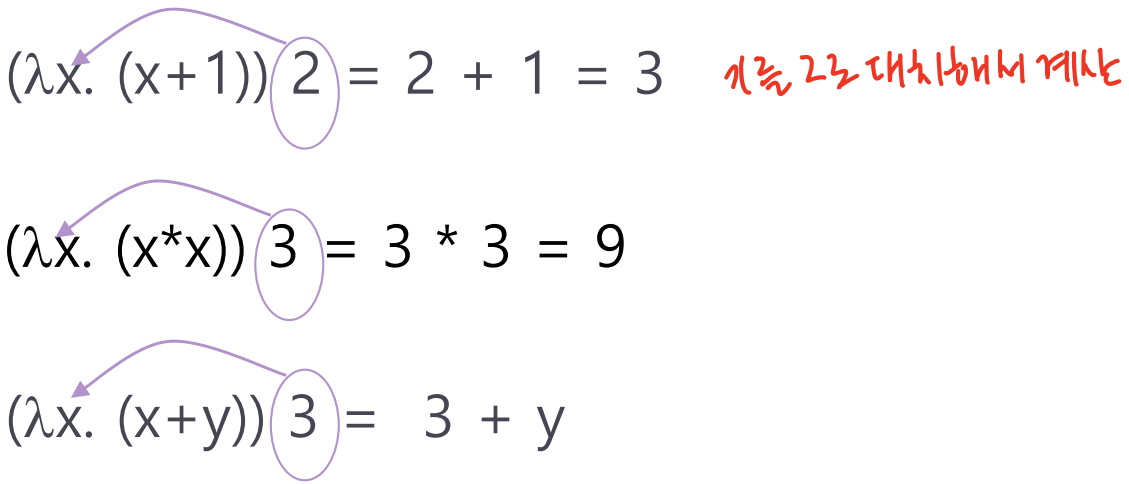
- 익명 함수

  λx. (x+1)

  λx. (x*x)

  λx. (x+y)

- 함수 적용



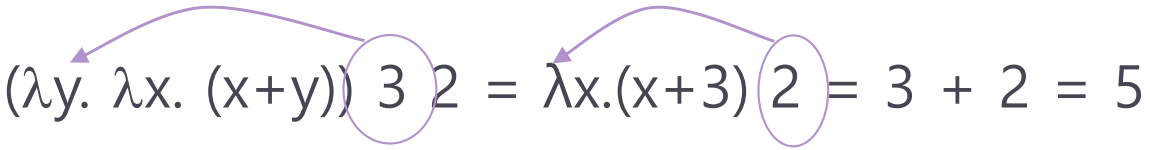
수식과 함수

- 익명 함수

  λy. λx. (x+y)

  λz. (x + 2*y + z)

- 함수 적용



축약(Reduction)

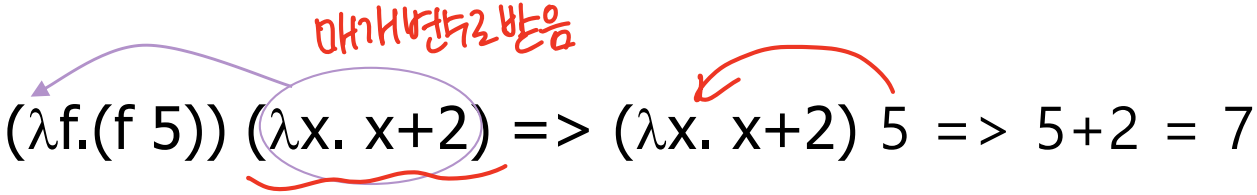
- β-축약이 기본 계산 규칙

  $(\lambda x.\ e_1)\ e_2 \quad => \quad e_1[e_2/x]$

- β-축약
  - 함수 적용을 매개변수에 대한 치환 연산의 결과로 대신하는 변환
  - e_1 내에 있는 매개변수 x를 e_2 로 대치한 후 e_1 를 계산한다.

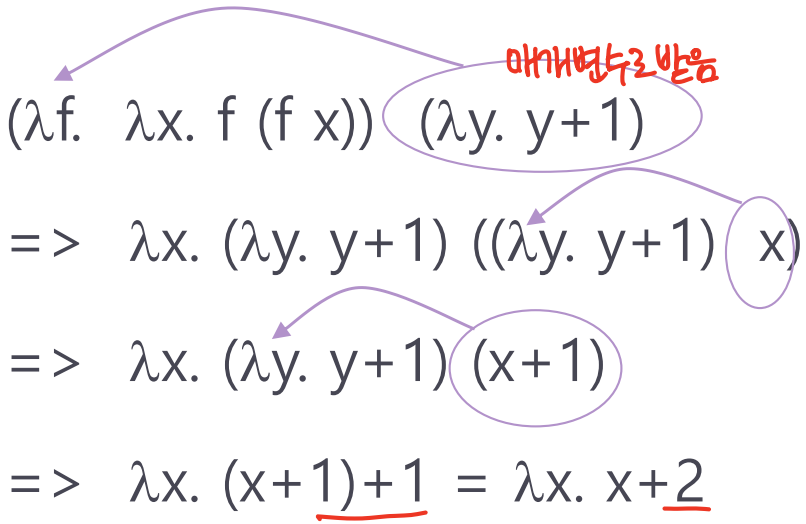
고차 함수

- 함수를 인자로 받거나 함수를 결과로 리턴하는 함수
- 예제 1



- 예제 2 : Given function f, return function f ∘ f

λf. λx. f (f x) λx. f (f x) = λx.(f (f x))



함수 선언과 구문적 편의

- 함수 선언과 호출

```
function f(x)
    return x+2
end;
f(5);
```

$(\lambda f.(f\ 5))\quad (\lambda x.\ x+2)$

block body declared function

- let 문장

$\text{let } x = e_1 \text{ in } e_2 \quad = \quad (\lambda x.\ e_2)\ e_1$

13.3 Scheme

Scheme

- Scheme
  - MIT에서 개발한 LISP 후속 함수형 언어
  - 리스트 자료구조
  - 프로그램과 데이터를 표현한다.
- 인터프리터
  - 리스트 형태의 식에 대해 읽기-평가-쓰기(read-evaluate-write)
  - 메타순환 해석기(metacircular interpreter)
  - 언어 자신을 이용하여 제작된 해석기
- 실행시간 메모리 관리
  - 메모리에 관한 모든 관리가 실행시간에 이루어진다.

식과 전위 표기법

- 식

  터미널X

  <expr> → <atom> | <list>

  <atom> → number | string | id

  <list> → '(' {<expr>} ')'

- 전위 표기법(prefix notation)
  - 모든 수식은 전위 표기법 형태로 표현

  연산자가 앞으로

- 예제

  왜? f(a,b) 이게 prefix 형태!

  > (+ 1 3)

  Value: 4

  > (* (+ 2 4) (- 6 2)

  Value: 24

함수 정의

- define를 이용하여 값 혹은 함수 정의

```
> (define (square x) (* x x))        // 함수 정의
> (square 5)                         // 함수 사용
 Value: 25

> (define pi 3.14159265)             // 값 정의 (생략)
> (define (circlearea r)             // 함수 정의
            (* pi (square r)))
> (circlearea 5)                     // 함수 사용
 Value: 78.53981625
```

익명 함수

- 람다식을 이용하여 익명 함수 표현
  - 예: (lambda (x) (* x x))

- 예

```
> ((lambda (x) (* x x)) 5)
25
```

- 익명 함수에 이름 정의

```
(define square (lambda (x) (* x x))
==
(define (square x) (* x x))
```

let 수식

- 형식

```
(let ( (이름1 식1)
          ...
        (이름n 식n) )
      식
)
```

Syntax 주의

- 예제

```
> (let ((x 7) (y 10))
       (+ x y))
Value: 17
```

술어 함수

- 부울 값을 반환하는 함수
  - 참: #t　　거짓: #f
- 술어 함수 예
  - =, <, >, >=, <=
  - even?
  - odd?
  - zero?
  - eq?

- 예

```
> (define x 11)
> (> x 11)
> (even? x)
```

조건 if

- 형식
  (if 조건 식1 식2)

- 예제

```
> (if (> 3 2) 'yes 'no)
Value: yes
> (define (test x)
      (if (>= x 70)
          (display "pass")
          (display "fail")))
> (test 75)
pass
```

재귀와 반복

- 반복
  - 재귀 함수를 이용한 반복

- 예제

```
> (define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))
  )

> (fact 3)
Value: 6
```

리스트

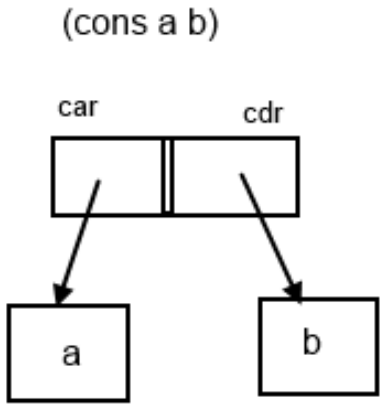
- 리스트

  '(a b c)　　'((a b) c d)　　'(1 3 5)　　'("hello" "world" "!")

- car 함수는 리스트의 첫번째 원소를 반환
  - (car '(a b c))　　=> a
  - (car '((a b) c d))　　=> (a b)
  - (car 'a)　　=> 오류(a 는 리스트가 아님).
  - (car '())　　=> 오류
- cdr 함수는 car가 제거된 나머지 리스트를 반환
  - (cdr '(a b c))　　=> (b c)
  - (cdr '((a b) c d))　　=> (c d)
  - (cdr 'a)　　=> 오류
  - (cdr '(a))　　=> ()

리스트 구성

- cons 리스트 구성자
  - (cons 'a '()) => (a)
  - (cons 'a '(b c)) => (a b c)
  - (cons '() '(a b)) => (() a b)
  - (cons '(a b) '(c d)) => ((a b) c d)



- append 함수
  - 두 개의 리스트를 접합한다(concatenate).
  - (append '(a b) '(c d)) => (a b c d)
  - (append '(1 2 3) '(4 5 6)) => (1 2 3 4 5 6)
  - (append '((a b) c) '(d (e f))) => ((a b) c d (e f))

```
(define (append list1 list2)
        (if (null? list1)
        list2
        (cons (car list1) (append (cdr list1) list2))))
```

고차 함수

- 함수를 매개변수로 받아 적용하는 함수
- mapcar 함수
  - 주어진 함수를 주어진 리스트의 각 원소에 적용하여 이 적용 결과의 리스트를 반환한다.

```
(define (mapcar fun list)
    (if (null? list)
        '()
        (cons (fun (car list)) (mapcar fun (cdr list)))
))
```

- mapcar 사용 예
  - (mapcar square '(3 4 2 6))
    Value: (9 16 5 36)
  - (mapcar (lambda(n) (* n n n)) '(3 4 2 6))
    Value: (27 64 8 216)

13.4 ML

ML

- 함수형 프로그래밍 언어
  - 프로그래밍 언어 분야의 핵심 연구 성과들을 잘 반영함.
  - 실용적인 언어로 많이 사용됨.
- 안전한 타입 시스템
  - 프로그램을 실행하는 중에 나올 수 있는 타입 오류를 실행하기 전에 미리 모두 찾아준다.
- Hindley-Milner 자동 타입 추론 시스템
  - 타입 표기를 생략한 경우에도 변수 혹은 수식들의 타입을 자동으로 추론할 수 있다.

ML의 특징

- 함수의 다형성(polymorphism)
  - 타입과 상관없이 실행할 수 있는 함수를 포괄적으로 정의 가능
- 대수적 자료형(algebraic data type)을 지원
  - 상위에서 자료구조를 표현할 수 있다.
- 값들의 패턴 매칭(pattern matching)
  - 간편하게 조건문을 만들 수 있다.
- 간단하고 강력한 예외처리(exception handling)
  - 프로그램의 실행흐름을 편리하게 기획할 수 있다.
- 메모리 재활용(garbage collection)
  - 자동으로 메모리를 관리한다.

식

- 값 선언문

  val 이름 = 식;

  val pi = 3.14159;

  ▪주의! 이렇게 선언된 이름은 값에 대한 이름이며 변수가 아니다.

- let 식

  let val 이름 = 식1 in 식2 end;

- 예

```
fun circlearea(r) =
let val pi = 3.14159
in
    pi * r * r
end;
```

자료구조

- ML의 자료구조
  - 리스트와 리스트 연산을 갖는다.
  - 열거형 타입, 배열, 레코드인 튜플 (tuple) 등

- 리스트는 대괄호를 사용

```
> [1,3,5];
val it = [1,3,5] : int list
```

- 리스트의 원소는 모두 같은 타입!

```
> [1,3,5.0];                    // 오류
```

튜플

- 튜플
  - 다른 타입의 데이터를 모으려면 리스트를 사용할 수 없음.
  - 튜플(tuple)을 사용해야 한다.

- 예

```
> (1,3,5.0);
val it = (1,3,5.0) : int * int * real
```

리스트 관련 연산자

- :: 연산자
  - Scheme의 cons처럼 첫번째 원소와 나머지 리스트로 새로운 리스트 구성

```
>1:: [3,5];
val it = [1,3,5] : int list
> 1::3::5::[];
val it = [1,3,5] : int list
```

- hd와 tl
  - Scheme의 car와 cdr에 해당하는 연산자

```
> hd [1,3,5];
val it = 1 : int
> tl [1,3,5];
val it = [3,5] : int list
```

제어 구조

- 조건식

  if E then 식1 else 식2

  - 조건을 나타내는 E의 값은 논리형이어야 하며
  - 두 식의 값은 같은 타입이어야 한다.

- 예

```
fun positive(n : int) : bool =
    if n > 0 then true
    else false;
```

함수 정의

- 함수 정의

  fun 함수이름 (매개변수) = 식;

  - 매개변수의 타입과 리턴타입을 명시할 수 있고
  - 생략해도 타입 추론에 의해서 함수의 타입을 결정한다.

- 예

  fun square (x : int) : int = x * x;

  fun square (x : int) = x * x;

  fun square (x) : int = x * x;

함수 호출

- 타입 없는 함수 정의

```
> fun square(x) = x * x ;
val square = fn : int -> int
```

- 함수 호출

```
> square(3);
val it = 9 : int
> square(3.0);
Error: operator and operand do not agree [tycon mismatch]
operator domain: int
operand: real
in expression:
square 3.0
```

실수 인자를 받는 square 함수

- 실수 인자를 받는 square 함수가 필요하면

```
fun square (x : real) : real = x * x;
fun square (x : real) = x * x;
fun square (x) : real = x * x;
```

  - ML은 함수의 중복정의(overloading)를 허용하지 않으므로
  - 이렇게 정수와 실수를 위한 square를 두 번 정의해도
  - square가 다형 함수로 정의된 것은 아님!
  - 단지 마지막으로 정의된 것만 유효하다.

익명 함수

- 람다식을 이용한 익명 함수 표현

  fn(매개변수) => 식; 또는

  fn 매개변수 => 식;

- 예

  > fn(x) => x * x; 혹은 fn x => x * x;

  val it = fn : int -> int

  > (fn(x) => x * x)(5);

  25

- 익명 함수에 이름을 정할 수 있다.

  > val square = fn(x) => x * x;

재귀 함수

- 재귀 함수(recursive function)
  - 함수에서 자신을 다시 호출하는 재귀 호출(recursive call) 방식으로 주어진 문제를 해결하도록 정의된 함수.

- 예

```
fun fact(n : int) : int =
    if n = 1 then 1
    else n * fact(n - 1);
```

패턴 매칭을 이용한 함수 정의

- 패턴 매칭을 이용한 함수 정의

```
fun id(<pattern1>) = <expr1>
  | id(<pattern2>) = <expr2>
      ...
  | id(<patternN>) = <exprN>
```

- fact 함수

```
fun fact(1 : int) : int = 1
    | fact(n : int) : int = n * fact(n - 1);
```

```
fun fact 1 = 1
  | fact n = n * fact(n-1);
```

- 함수 호출

```
fact(4);
val it = 24 : int
```

리스트 append 함수

- append 함수
  - 두 개의 리스트를 접합한다(concatenate).

```
> fun append([], L) = L
      | append(h::t, L) = h :: append(t, L);
val append = fn : 'a list * 'a list -> 'a list
```

함수 정의 방법

- 첫번째 방법

```
> fun plus(x, y) = x + y;
val plus = fn : int * int -> int
> plus(3,5);
val it = 8 : int
```

- 두 번째 커링 함수 정의

```
> fun plus x y = x + y;
val plus = fn : int -> int -> int
> plus 3 5;
val it = 8 : int
```

커링 함수

- 커링 함수 사용

```
> plus 3;
val it = fn : int -> int
```

  - 따라서 이 결과 함수를 다시 값에 적용할 수 있다.

```
> plus 3 5;
val it = 8 : int
```

  - 결과 함수에 이름을 주고 이를 다시 값에 적용하는 것도 가능.

```
> val plus3 = plus 3;
val it = fn : int -> int
> plus3 5
val it = 8 : int
```

커링 함수

- 커링(Currying) 함수
  - n개의 매개변수를 받는 하나의 함수를
  - 단일 매개변수를 받는 n개 함수들의 열로 만드는 것을 말한다.

- 재귀 함수도 커링 함수로 정의할 수 있음

```
> fun exp x 0 = 1
      | exp x y = x * exp x (y-1);
val exp = fn : int -> int -> int
> exp 3 5;
exp val it = 243 : int
```

고차 함수

- 고차 함수(higher-order function)
  - 함수를 인자로 받아 적용하는 함수!
  - 함수가 일등급 값이므로 함수의 인자로 함수를 받아 이를 적용할 수 있다.

- map 함수

```
fun map f [] = []
    | map f (x :: xs) = f(x) :: map(f)(xs);
```

- 사용 예

```
> map(square) ([2, 4, 6]);
val it = [4,16,36] : int list
> map (fn x => x * x * x) ([2, 4, 6]);
val it = [8,64,216] : int list
```