

SNU 4190.210 프로그래밍 원리(Principles of Programming) Part I

Prof. Chung-Kil Hur

School of Computer Science & Engineering

SNU 4190.210 ©Kwangkeun Yi

차례

- 1 프로그래밍 기본부품과 조합 (elements & compound)
- 2 이름짓기 (binding, delclaration, definition)
- 3 재귀와 고차함수 (recursion & higher-order functions)
- 4 타입으로 정리하기 (types & typeful programming)
- 5 프로그램의 계산 복잡도 (program complexity)
- 6 맞는 프로그램인지 확인하기 (program correctness)

SNU 4190.210 ©Kwangkeun Yi

다음

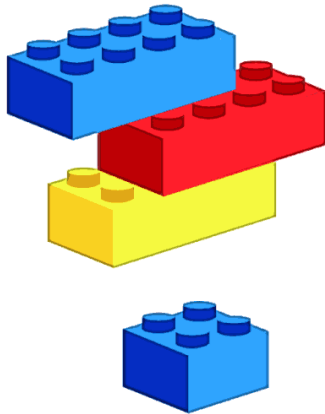
- 1 프로그래밍 기본부품과 조합 (elements & compound)
- 2 이름짓기 (binding, declaration, definition)
- 3 재귀와 고차함수 (recursion & higher-order functions)
- 4 타입으로 정리하기 (types & typeful programming)
- 5 프로그램의 계산 복잡도 (program complexity)
- 6 맞는 프로그램인지 확인하기 (program correctness)

프로그램 구성에 필요한 요소 (Elements of Programming)

- ▶ 기본부품(primitives)
- ▶ 조합하는 방법(means of constructing compound)
- ▶ 프로그램 실행과정의 이해(rule of evaluation)
- ▶ 타입으로 정리하는 방법(types)
- ▶ 속내용을 감추는 방법(means of abstraction)

기본부품의 반복된 조합 (Combination of Primitives)

(pictures from Google search)



Difference: programs can be executed

기본 부품(primitives)

기본적으로 제공됨. 상수(constant)라고도 불림.

type	elements	operators
\mathbb{N}, \mathbb{R}	0, -1, 1.2, -1.2e2	+, *, /, =, <=, ...
\mathbb{B}	#t, #f	and, or, not, ...
String	"snow", "12@it"	substr, strconcat, ...
Symbol	'snow, '12@it	
Unit	()	

- ▶ Base types: \mathbb{N} , \mathbb{R} , \mathbb{B} , String, Symbol
- ▶ 실행(evaluation, semantics): -1.2e2 is -1.2×10^2 , #t is true, + is +, 'snow is a symbol named "snow", etc

식을 조합하는 방법 (Ways to Construct Programs)

$P ::= E$	program
$E ::= c$	constant
x	name
$(\text{if } E E E)$	conditional
$(\text{cons } E E)$	pair
$(\text{car } E)$	selection
$(\text{cdr } E)$	selection
$(\text{lambda } (x^*) E)$	function
$(E E^*)$	application

- ▶ 재귀적(inductive, recursive):
 - ▶ One can compose infinitely many programs.
 - ▶ Arbitrary expressions can be plugged into an expression.
- ▶ When composing a program, one has to understand how the program will be evaluated (semantics).

SNU 4190.210 ©Kwangkeun Yi

프로그램 식의 실행과정 (Evaluation Process)

- ▶ 주어진 프로그램 식을 읽고(read)
- ▶ 그 식을 계산하고(evaluate)
 - ▶ 계산중에 컴퓨터 메모리와 시간을 소모 (consume time & memory)
 - ▶ 계산중에 입출력이 있으면 입출력을 수행 (input & output)
- ▶ 최종 계산 결과가 있으면 화면에 프린트한다(print)

주의:

- ▶ 식의 실행 규칙(rule of evaluation, semantics): clearly defined
- ▶ 프로그래머는 이것을 이해해야 의도한 프로그램을 작성할 수 있음 (programmer should understand the semantics)
- ▶ 제대로 실행될 수 없는(오류있는) 멀쩡한 식들이 많음 (Many well-formed programs are erroneous)

SNU 4190.210 ©Kwangkeun Yi

식의 실행규칙(rule of evaluation, semantics)

실행 규칙. 각 식의 종류에 따라서 (Each case has its own rule):

- ▶ c 일때:
- ▶ x 일때:
- ▶ $(\text{if } E E E)$ 일때:
- ▶ $(\text{cons } E E)$ 일때:
- ▶ $(\text{car } E)$ 일때:
- ▶ $(\text{cdr } E)$ 일때:
- ▶ $(\text{lambda } (x^*) E)$ 일때:
- ▶ $(E E^*)$ 일때:

주의:

- ▶ 생긴게 옳다고 모두 제대로 실행되는 게 아님
(Well-formed programs may go wrong)
- ▶ 부품식들의 계산결과의 타입이 맞아야 (Well-typed programs do not go wrong)
- ▶ 이름의 사용 (use of names), 이름의 유효범위(scope)

SNU 4190.210 ©Kwangkeun Yi

프로그램식 조합방식의 원리 (Principle about Program Components)



모든 프로그래밍 언어에는 각 타입마다 그 타입의 값을 만드는 식과 사용하는 식을 구성하는 방법이 제공된다.

[For each type, there are ways to make values of the type (introduction) and ways to use them (elimination).]



SNU 4190.210 ©Kwangkeun Yi

이 원리를 확인해보면 (Let's check this principle)

타입 τ	만드는 식 (intro)	사용하는 식 (elim)
기본타입 ι	c	$+$, $*$, $=$, and, substr, etc
곱타입 $\tau \times \tau$	$(\text{cons } E \ E)$	$(\text{car } E)$, $(\text{cdr } E)$
함수타입 $\tau \rightarrow \tau$	$(\text{lambda } (x^*) \ E)$	$(E \ E^*)$

다음

- 1 프로그래밍 기본부품과 조합 (elements & compound)
- 2 이름짓기 (binding, delclaration, definition)
- 3 재귀와 고차함수 (recursion & higher-order functions)
- 4 타입으로 정리하기 (types & typeful programming)
- 5 프로그램의 계산 복잡도 (program complexity)
- 6 맞는 프로그램인지 확인하기 (program correctness)

Naming (binding, declaration, definition)



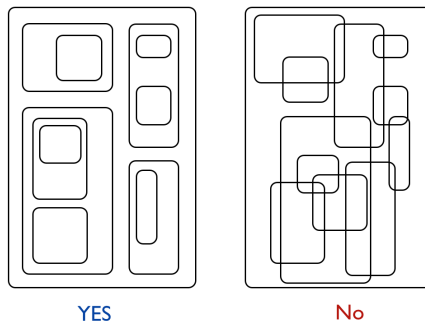
이름 짓기는 속내용감추기(abstraction)의 첫 스텝: 이름을 지으면 지칭하는 대상(속내용) 대신에 그 이름을 사용
(Naming is a simple way of abstraction: One can hide details by giving it a name)

- ▶ 이름 지을 수 있는 대상 (those that can be named):
 - ▶ 프로그램에서 다룰 수 있는 모든 값 (all values)
- ▶ 이름의 유효범위가 한정됨 (The scope of a name is given). 따라서,
 - ▶ 이름 재사용 가능 (reuse a name)
 - ▶ 전체 프로그램의 모든 이름을 외을 필요없음 (do not need to know all names)
 - ▶ 이름이 필요한 곳에만 알려짐 (a name is defined only where it is needed)
- ▶ 이름의 유효범위는 쉽게 결정됨 (The scope of a name is easily determined)

SNU 4190.210 ©Kwangkeun Yi

이름의 유효범위 결정 (how to determine the scope)

프로그램 텍스트에서 쉽게 결정됨(lexical scoping) [The scope of a name is easily determined from the program text]



이런 간단한 유효범위는 수리논술의 2000년 전통 (The idea originates from Math):

Theorem The intersection of all addition-closed sets is addition-closed.

Proof Let S be the intersection set. Let x and y be elements of S . Because x and y are elements of ... hence in S . \square

SNU 4190.210 ©Kwangkeun Yi

이름짓기(binding, declaration, definition)

▶ 식에서 이름짓기

$E ::= \dots$	예전 것들
$(\text{let } ((x E)^+) E)$	x 의 정의
$(\text{letrec } ((x E)^+) E)$	x 의 재귀 정의

▶ 프로그램에서 이름짓기

$P ::= E$	계산식
$(\text{define } x E)^* E$	이름정의 후 계산식

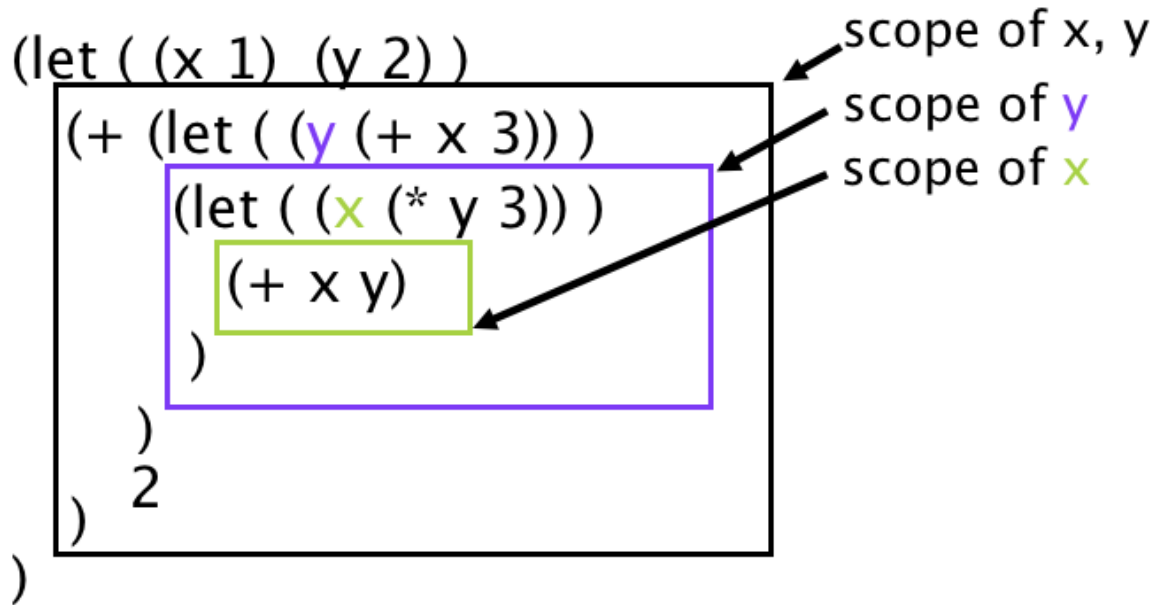
이름짓기의 실행규칙(rule of evaluation, semantics)

- ▶ $(\text{let } ((x E)) E)$
- ▶ $(\text{letrec } ((x E)) E)$
- ▶ $(\text{define } x E) E$

주의 (Note):

- ▶ 생긴게 옳다고 모두 제대로 실행되는 게 아님 (Well-formed programs may go wrong)
- ▶ 부품식들의 계산결과와 타입이 맞아야 (Well-typed programs do not go wrong)
- ▶ 이름의 유효범위(scope)
- ▶ 환경(environment): 이름과 그 대상(값)의 목록표 (table for names and their associated values)

이름의 유효범위(scope) 예



SNU 4190.210 ©Kwangkeun Yi

이름짓기 + 사용하는의 실행과정(semantics)

컴퓨터는 프로그램 식을 실행할 때 (When evaluating an expression)

- ▶ 이름과 그 대상의 목록표를 관리 (managing a table for names and their associated values)
- ▶ 그러한 목록표를 환경이라고 함 (Such a table is called an environment)

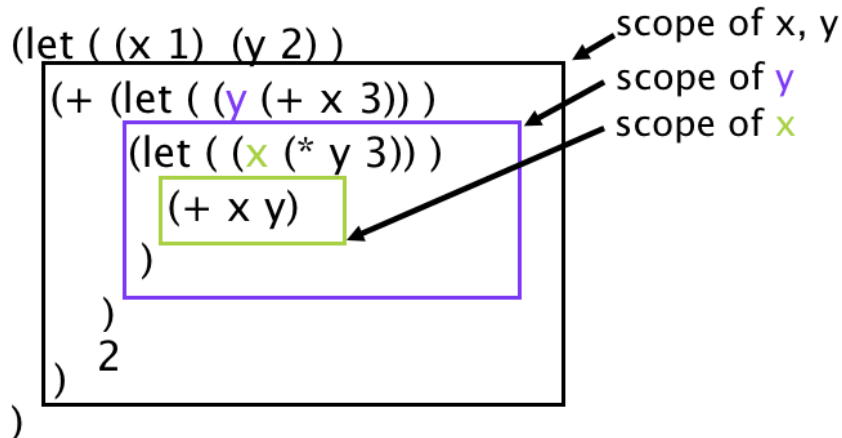
a	1
b	2
env	(('a 1) ('b 2))
f	(lambda (x) (+ x 1))

SNU 4190.210 ©Kwangkeun Yi

이름짓기 + 사용하기의 실행과정(semantics)

환경 관리 (Managing environments)

- ▶ 환경 만들기: 이름이 지어지면 (Make an environment when names are defined)
- ▶ 환경 참조하기: 이름이 나타나면 (Use the environment when the names are used)
- ▶ 환경 폐기하기: 유효범위가 끝나면 (Delete the environment when the scope ends)



SNU 4190.210 ©Kwangkeun Yi

여러개 한꺼번에 이름짓기: 실행의미 (Multiple names at once)

- ▶ $(\text{let } ((x_1 E_1) (x_2 E_2)) E)$
- ▶ $(\text{letrec } ((x_1 E_1) (x_2 E_2)) E)$
- ▶ $(\text{define } x_1 E_1) (\text{define } x_2 E_2) E$

SNU 4190.210 ©Kwangkeun Yi

설탕구조(syntactic sugar)

편리를 위해서 제공; 지금까지것들로 구성가능; 반드시 필요는 없다 (Just convenient notations for “expressions”):

`list`, `cond`, `let`, `define`은 설탕

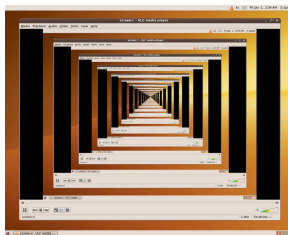
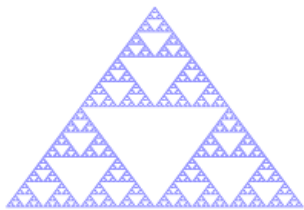
<code>(list E^*)</code>	=	<code>(cons ...)</code>
<code>(cond ($E E'$) (else E''))</code>	=	<code>(if ...)</code>
<code>(let (($x E$)) E')</code>	=	<code>((lambda ...) ...)</code>
<code>(let (($x_1 E_1$) ($x_2 E_2$)) E)</code>	=	<code>((lambda ...) ...)</code>
<code>(define $x E$) E'</code>	=	<code>(letrec ...)</code>
<code>(define $x E$) (define $y E'$) E''</code>	=	<code>(letrec ...)</code>
<code>(define ($f x$) E)</code>	=	<code>(define ...)</code>
<code>(begin $E E'$)</code>	=	<code>((lambda ...) ...)</code>

다음

- 1 프로그래밍 기본부품과 조합 (elements & compound)
- 2 이름짓기 (binding, delclaration, definition)
- 3 재귀와 고차함수 (recursion & higher-order functions)
- 4 타입으로 정리하기 (types & typeful programming)
- 5 프로그램의 계산 복잡도 (program complexity)
- 6 맞는 프로그램인지 확인하기 (program correctness)

재귀(recursion): 되돌기, 같은 일의 반복

- ▶ 예) 재귀하고있는 그림들:



- ▶ 예) 재귀하고있는 표기법:

$$E ::= c$$
$$| x$$
$$| (\text{if } E E E)$$
$$| (\text{cons } E E)$$

- ▶ 예) 재귀하고있는 정의:

$$a_0 = 1, \quad a_{n+1} = a_n + 2 \quad (n \in \mathbb{N})$$
$$X = 1 \leftrightarrow X$$

재귀함수(recursive function)의 정의

- ▶ 함수만 재귀적으로 정의가능 (대부분의 언어) (왜?)
[Only functions can be defined recursively in most langs (why?)]

```
(define fac
  (lambda (n) (if (= n 0) 1
                  (* n (fac (- n 1)))))
))
```

- ▶ 임의의 값을 재귀적으로 정의? 그 값 계산이 무한할 수 있음 (possible in Haskell) [Recursively defined data can be infinitely large]

```
(define x (+ 1 x))
(define K (cons 1 K))
(define Y (cons 1 (add1 Y)))
```

재귀함수의 실행과정

```
(define (fac n)
  (if (= n 0) 1
      (* n (fac (- n 1)))))

(fac 4)
⇒(* 4 (fac 3))
⇒(* 4 (* 3 (fac 2)))
⇒(* 4 (* 3 (* 2 (fac 1))))
⇒(* 4 (* 3 (* 2 (* 1 (fac 0))))))
⇒(* 4 (* 3 (* 2 (* 1 1))))
⇒(* 4 (* 3 (* 2 1)))
⇒(* 4 (* 3 2))
⇒(* 4 6)
⇒24
```

- ▶ 누적됨: 재귀호출을 마치고 계속해야 할 일들이
 - ▶ 함수호출때 호출 마치고 계속해야 할 일(continuation)을 기억해야
- ▶ 현대기술은 재귀호출때 누적 안되도록 자동변환
 - ▶ 끝재귀(tail recursion) 변환

SNU 4190.210 ©Kwangkeun Yi

끝재귀(tail recursion) 변환

```
(define (fac n)
  (if (= n 0) 1
      (* n (fac (- n 1)))))
))
```

```
(define (fac n)
  (define (fac-aux m r)
    (if (= m 0) r
        (fac-aux (- m 1) (* m r))))
  (fac-aux n 1))
```

SNU 4190.210 ©Kwangkeun Yi

끝재귀함수(tail-recursive ftn)의 실행 과정

(fac 4)

⇒ (fac-aux 4 1)

⇒ (fac-aux 3 (* 4 1))

⇒ (fac-aux 3 4)

⇒ (fac-aux 2 (* 3 4))

⇒ (fac-aux 2 12)

⇒ (fac-aux 1 (* 2 12))

⇒ (fac-aux 1 24)

⇒ (fac-aux 0 (* 1 24))

⇒ (fac-aux 0 24)

⇒ 24

- ▶ 할일을 (하고) 재귀호출 변수로 전달
- ▶ 재귀호출 마치고 할 일이 누적되지 않음

고차함수(higher-order function)

- ▶ 함수가 인자로

$$\sum_{n=a}^b f(n) = f(a) + \cdots + f(b)$$

- ▶ 함수가 결과로

$$\frac{d}{dx} f(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x - \epsilon)}{\epsilon}$$

현대 프로그래밍에서

- ▶ 모두 지원되는(Scala, Python, Lua, JavaScript, Clojure, Scheme, ML, C#, F#등)
- ▶ 과거에는 지원되지 못했던

고차함수는 일상에서 흔하다

- ▶ 함수가 인자로
 - ▶ 요리사(함수)는 요리법(함수)과 재료를 받아서...
 - ▶ 댄서(함수)는 리듬있게움직이는법(함수)과 음악을 받아서...
 - ▶ 컴퓨터(함수)는 프로그램(함수)과 입력을 받아서...
- ▶ 함수가 결과로
 - ▶ 요리학교(함수)는 요리사(함수)를 만들어내고
 - ▶ 댄스동아리(함수)는 댄서(함수)를 만들어내고
 - ▶ 컴퓨터공장(함수)은 컴퓨터를(함수) 만들어내고

고차함수의 쓸모

고수준으로 일반화된 함수를 정의할 수 있다

```
(define (sigma lower upper)
  (lambda (f)
    (define (loop n)
      (if (> n upper) 0
          (+ (f n) (loop (+ n 1)))))
      (loop lower)
    ))
```

```
(define one-to-million (sigma 1 1000000))
(one-to-million (lambda (n) (* n n)))
(one-to-million (lambda (n) (+ n 2)))
```

고차함수의 쓸모

고수준으로 일반화된 함수를 정의할 수 있다

```
(define (sum lower upper f)
  (if (> lower upper) 0
      (+ (f lower) (sum (+ lower 1) upper f))
      ))
```

```
(define (generic-sum lower upper f larger base op inc)
  (if (larger lower upper) base
      (op (f lower)
          (generic-sum (inc lower) upper f larger base op inc))
      ))
```

```
(sum 1 10 (lambda (n) n))
```

```
(sum 10 100 (lambda (n) (+ n 1)))
```

```
(generic-sum 1 10 (lambda (n) n) > -1 + (lambda (n) (+ 2 n)))
```

```
(generic-sum "a" "z" (lambda (n) n) order "" concat alpha-next)
```

SNU 4190.210 ©Kwangkeun Yi

다음

- 1 프로그래밍 기본부품과 조합 (elements & compound)
- 2 이름짓기 (binding, declaration, definition)
- 3 재귀와 고차함수 (recursion & higher-order functions)
- 4 타입으로 정리하기 (types & typeful programming)
- 5 프로그램의 계산 복잡도 (program complexity)
- 6 맞는 프로그램인지 확인하기 (program correctness)

SNU 4190.210 ©Kwangkeun Yi

타입(type)



타입(type)은 프로그램이 계산하는 값들의 집합을 분류해서 요약하는 데 사용하는 “언어”이다. 타입으로 분류요약하는 방식은 대형 프로그램을 실수없이 구성하는 데 효과적이다.

Type is a “language” for describing a set of values that are used in programs. Abstracting a program as a type is an effective method to reduce errors in writing large programs.

- ▶ 타입(type)은 가이드다 [Type is a guide]
 - ▶ 프로그램의 실행안전성을 확인하는 [for removing runtime errors]
 - ▶ 새로운 종류의 데이터값을 구성하는 [for describing a new data structure]

사용하는 타입들(types)

$\tau ::= \iota$	primitive type
$\tau \times \tau$	pair(product) type
$\tau + \tau$	or(sum) type
$\tau \rightarrow \tau$	ftn type, single param
$\tau * \dots * \tau \rightarrow \tau$	ftn type, multi params
\top	any type
t	user-defined type's name
τt	user-defined type's name, with param
$\iota ::= int \mid real \mid bool \mid string \mid symbol \mid unit$	

고차함수 타입 [higher-order types]

고차함수 타입 예:

$int * int * (int \rightarrow int) \rightarrow int$
 $(real \rightarrow real) \rightarrow (real \rightarrow real)$
 $int * (int \rightarrow int) \rightarrow int$
 $int \rightarrow (int \rightarrow int) \rightarrow int$
 $int \times (int \rightarrow int) \rightarrow int$
 $(int \rightarrow int) \rightarrow int\ list \rightarrow int$
 $(int \rightarrow int) \times int\ list \rightarrow int$
 $money \rightarrow (year \rightarrow car\ list)$

타입을 상상하며 식을 구성하기 [write programs with types in mind]

- ▶ $c : \iota$
- ▶ $x : \tau$
- ▶ $(\text{if } E:\mathbb{B} \ E':\tau \ E'':\tau) : \tau$
- ▶ $(\text{lambda } (x:\tau) \ E:\tau') : \tau \rightarrow \tau'$
 - ▶ $x : \tau$ 임을 E 를 구성할 때 기억
- ▶ $(E:\tau' \rightarrow \tau \ E':\tau') : \tau$
- ▶ $(\text{cons } E:\tau \ E':\tau') : \tau \times \tau'$
- ▶ $(\text{car } E:\tau \times \tau') : \tau$
- ▶ $(\text{cdr } E:\tau \times \tau') : \tau'$

타입을 상상하며 식을 구성하기 [write programs with types in mind]

- ▶ $c : \iota$
- ▶ $x : \tau$
- ▶ $(\text{if } E:\mathbb{B} \ E':\tau \ E'':\tau) : \tau$
- ▶ $(\text{lambda } (x:\tau) \ E:\tau') : \tau \rightarrow \tau'$
 - ▶ $x : \tau$ 임을 E 를 구성할 때 기억
- ▶ $(E:\tau' \rightarrow \tau \ E':\tau') : \tau$
- ▶ $(\text{cons } E:\tau \ E':\tau') : \tau \times \tau'$
- ▶ $(\text{car } E:\tau \times \tau') : \tau$
- ▶ $(\text{cdr } E:\tau \times \tau') : \tau'$

SNU 4190.210 ©Kwangkeun Yi

타입을 상상하며 식을 구성하기 [write programs with types in mind]

- ▶ $(\text{let } ((x_1:\tau_1 \ E_1:\tau_1) \ (x_2:\tau_2 \ E_2:\tau_2)) \ E:\tau)$
 - ▶ $x_1 : \tau_1$ 이고 $x_2 : \tau_2$ 임을 E 를 구성할 때 기억
- ▶ $(\text{letrec } ((x_1:\tau_1 \ E_1:\tau_1) \ (x_2:\tau_2 \ E_2:\tau_2)) \ E:\tau)$
 - ▶ $x_1 : \tau_1$ 이고 $x_2 : \tau_2$ 임을 E_1, E_2, E 를 구성할 때 기억
- ▶ $(\text{define } x_1:\tau_1 \ E_1:\tau_1) \ (\text{define } x_2:\tau_2 \ E_2:\tau_2) \ E:\tau)$
 - ▶ $x_1 : \tau_1$ 이고 $x_2 : \tau_2$ 임을 E_1, E_2, E 를 구성할 때 기억

SNU 4190.210 ©Kwangkeun Yi

타입으로 프로그램을 정리/검산하기 [Validate a program using types]

```
(define (fac n)
  (if (= n 0) 1
      (* n (fac (- n 1)))
  ))

(define (fibonacci n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fibonacci (- n 1))
                  (fibonacci (- n 2)))))
  ))

(define (bar a b c)
  (if (= b 0) c
      (bar (+ a 1) (- b 1) (* a b))
  ))
```

SNU 4190.210 ©Kwangkeun Yi

타입으로 프로그램을 정리/검산하기 [Validate a program using types]

```
{= : int * int -> bool}
{* : int * int -> int}
{- : int * int -> int}

{(define (fac {n : int})
  {(if {(= {n : int} {0 : int}) : bool}
      {1 : int}
      {(* {n : int}
          {(fac {(- {n:int} {1:int}) : int}) : int}
      ) : int}
  ) : int}
) : int -> int}
```

SNU 4190.210 ©Kwangkeun Yi

Type checking rule for lists

$$\frac{}{()' : \tau \text{ list}}$$
$$\frac{a : \tau \quad l : \tau \text{ list}}{(\text{cons } a \ l) : \tau \text{ list}}$$

타입으로 프로그램을 정리/검산하기 [Validate a program using types]

```
(define (map-reduce f l op init)
  (reduce (map f l) op init))

(define (map f l)
  (if (null? l) '()
      (cons (f (car l)) (map f (cdr l)))
  ))

(define (reduce l op init)
  (if (null? l) init
      (op (car l) (reduce (cdr l) op init))
  ))

(define (word-count pages) (map-reduce wc pages + 0))
(define (make-dictionary pages)
  (map-reduce mw (words pages) merge ()))
```

다음

- 1 프로그래밍 기본부품과 조합 (elements & compound)
- 2 이름짓기 (binding, declaration, definition)
- 3 재귀와 고차함수 (recursion & higher-order functions)
- 4 타입으로 정리하기 (types & typeful programming)
- 5 프로그램의 계산 복잡도 (program complexity)
- 6 맞는 프로그램인지 확인하기 (program correctness)

계산복잡도(complexity)

프로그램 실행 비용의 증가정도(order of growth)

- ▶ 계산비용 [cost] = 시간과 메모리 [time & memory]
- ▶ 증가정도 [growth] = 입력 크기에 대한 함수로 [function on input size], 단
 - ▶ 관심: 입력이 커지면 결국 어떻게 될지(asymptotic complexity)
- ▶ “계산복잡도(complexity, order of growth)가 $\Theta(f(n))$ 이다”(n은 입력의 크기), 만일 그 복잡도가 $f(n)$ 으로 샌드위치될때 [if $f(n)$ satisfies the following]:

$$\Theta(f(n)) : k_1 \times f(n) \leq \bullet \leq k_2 \times f(n)$$

$$O(f(n)) : \bullet \leq k_2 \times f(n)$$

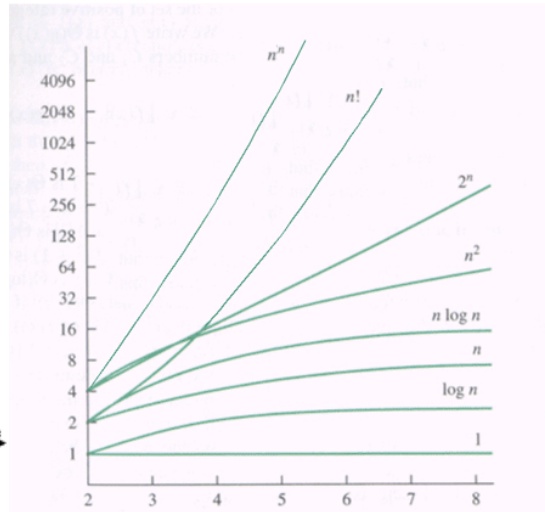
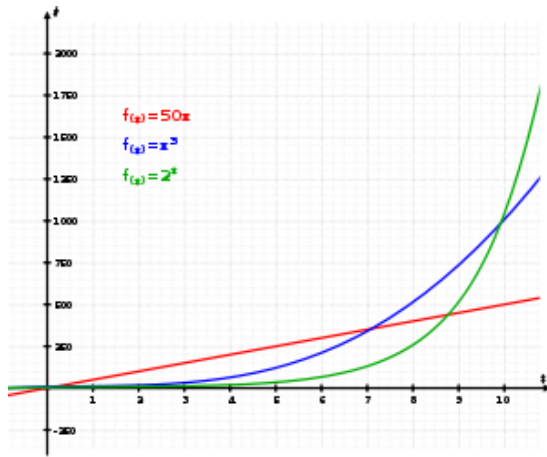
$$\Omega(f(n)) : k_1 \times f(n) \leq \bullet$$

(k_1, k_2 는 n 과 무관한 양의 상수 [k_1, k_2 are fixed constants])

- ▶ $n^2, 10000 \times n^2, 3 \times n^2 + 10000 \times n$ 은 모두 $\Theta(n^2)$

계산복잡도(complexity)

(pictures from Google search)



SNU 4190.210 ©Kwangkeun Yi

계산복잡도(complexity)

- ▶ (fac n): 시간복잡도[time complexity] $\Theta(n)$, [space complexity]메모리복잡도 $\Theta(n)$.
- ▶ (exp b n):
 - ▶ $O(n)$ 로 구현가능 [yes]
 - ▶ $O(\log n)$ 로 구현가능 [yes]
- ▶ (SAT formula):
 - ▶ $O(2^n)$ 로 구현가능 [yes]
 - ▶ $O(\text{poly}(n))$ 로 구현가능? 누구도모름 [not known]
- ▶ (diophantine eqn):
 - ▶ $O(2^n)$ 로 구현가능?
 - ▶ $O(n^n)$ 로 구현가능?
 - ▶ Is it decidable? not known until 1970, then “no”

SNU 4190.210 ©Kwangkeun Yi

다음

- 1 프로그래밍 기본부품과 조합 (elements & compound)
- 2 이름짓기 (binding, delclaration, definition)
- 3 재귀와 고차함수 (recursion & higher-order functions)
- 4 타입으로 정리하기 (types & typeful programming)
- 5 프로그램의 계산 복잡도 (program complexity)
- 6 맞는 프로그램인지 확인하기 (program correctness)

맞는 프로그램인지 확인하기 [program correctness]

프로그램을 돌리기 전에 (static analysis) [before execution]

- ▶ 분석 검증 후 프로그램 제출/출시/탑재 [execute after static analysis]
- ▶ 다른 공학분야와 동일 [the same in other areas of engineering]:
 - ▶ 기계/전기/공정/건축설계 분석 검증 후 제작/설비/건설 [build after analysis in mechanical/electrical/architecture engineering]
- ▶ 일상과 동일 [the same in our life]:
 - ▶ 입시/면접, 사주/공합, 클럽기도
 - ▶ 검증 후 실행

4190.210에서는 간단한 기술만

검증해야 할 성질들 (properties to verify)

- ▶ 제대로 생겼는가? 자동검증 [well formed? automatic]
- ▶ 타입에 맞게 실행될 것인가? [well typed?]
 - ▶ 직접검증(Scheme, C, JavaScript, etc) [manual]
 - ▶ 자동검증(ML, Scala, Haskell, Java, Python, etc) [automatic]
- ▶ 4190.210: 항상 끝나는가: 직접검증, 비교적 용이 [terminate?: manual, not so hard]
- ▶ 내가 바라는 계산을 하는가: 어려움 [fully correct?: very hard]

My main research topic.

You can learn some from the “Software Foundations” course that I will teach in 2015 Fall.

확인: 모든 입력에 대해서 정의되었는가? [does not crash?]

- ▶ 타입에 맞게 실행될 것이 확인된 경우 [just need to check whether the program is well typed]

We will learn more complex types for user-defined data structures later.

확인: 항상 끝나는가? [always terminate?]

그렇다, 만일 [If]:

- ▶ 반복 될 때 뭔가가 계속 “줄어들고” [something “decreases”]
- ▶ 그 줄어듬의 “끝이 있다” 면 [the decreasing has an end].

즉, 재귀함수의 경우, 만일 [For example, in a recursive function, if]:

- ▶ 재귀 호출마다 인자가 “줄어들고” [an argument decreases at every recursive call]
- ▶ 그 줄어듬의 “끝이 있다” 면 [the decreases has an end].

끝나는 재귀함수인지 확인하기 [terminate?]

```
(define (fac n)
  (if (= n 0) 1
      (* n (fac (- n 1)))))
))
```

- ▶ 음이아닌 정수만 입력으로 받는다면 [if $n \geq 0$],
- ▶ 재귀호출마다 원래 n 보다 줄고 있고 “ $n-1$ ” [$n - 1 < n$ at every recursive call],
- ▶ 끝이 있다(“(= n 0) 1”) [there is an end].

끝나는 재귀함수인지 확인하기 [terminate?]

```
(define (fibonacci n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fibonacci (- n 1))
                  (fibonacci (- n 2))))))
```

일반방법: 끝나는 재귀함수인지 확인하기 [terminate?]

아래와 같은 재귀함수 $f : A \rightarrow B$ 를 생각하자(w.l.o.g.)

```
(define (f x) ... (f e1)... (f e2)...)
```

재귀함수 인자들의 집합 A 에서 [in the set A]

- ▶ 원소들 간의 줄어드는 순서 $>$ 를 찾으라, 아래와 같은 [find $>$ such that]:
- ▶ 재귀호출 인자식(e_1 와 e_2)의 값이 원래 인자(x)보다 $>$ 인("줄어드는"), 그리고 [$x > e_1$ and $x > e_2$]
- ▶ 집합 A 에서 $>$ -순서대로 원소를 줄세우면 항상 유한한(well-founded) [$>$ has no infinite decreasing chain (i.e., well-founded) in A].

끝나는 재귀함수인지 확인하기 [terminate?]

```
(define (bar a b c)
  (if (= b 0) c
      (bar (+ a 1) (- b 1) (* a b))
  ))
```

- ▶ $\mathbb{N} * \mathbb{N} * \mathbb{N}$ 에서 줄어드는 순서 $>$ 는? [what is $>$?]
- ▶ 그래서 그 $>$ -순서가 항상 유한번에 바닥에 닿는(well-founded)? [is $>$ well-founded?]

그런 순서 $>$ 는?

끝나는 재귀함수인지 확인하기 [terminate?]

```
(define (map-reduce f l op init)
  (reduce (map f l) op init))
```

```
(define (map f l)
  (if (null? l) ()
      (cons (f (car l)) (map f (cdr l)))
  ))
```

```
(define (reduce l op init)
  (if (null? l) init
      (op (car l) (reduce (cdr l) op init))
  ))
```

```
(define (word-count pages) (map-reduce wc pages + 0))
(define (make-dictionary pages) (map-reduce mw (words page
```

끝나는 재귀함수인지 확인하기 [terminate?]

```
(define (sum lower upper f)
  (if (> lower upper) 0
      (+ (f lower) (sum (+ lower 1) upper f))
      )))
```

끝나는 재귀함수인지 확인하기 [terminate?]

```
(define (sigma lower upper)
  (lambda (f)
    (define (loop n)
      (if (> n upper) 0
          (+ (f n) (loop (+ n 1)))))
    (loop upper)
    )))
```

끝나는 재귀함수인지 확인하기 [terminate?]

```
(define (ackermann m n)
  (cond
    ((= m 0) (+ n 1))
    ((= n 0) (ackermann (- m 1) 1))
    (else (ackermann (- m 1) (ackermann m (- n 1)))))
  ))
```