

SNU 4190.210 프로그래밍 원리(Principles of Programming) Part I

Prof. Kwangkeun Yi

School of Computer Science & Engineering

차례

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

다음

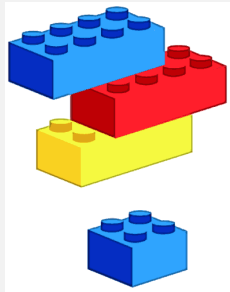
- 1 프로그래밍 기본부품과 조합 (elements & compound)
- 2 이름짓기 (binding, declaration, definition)
- 3 재귀와 고차함수 (recursion & higher-order functions)
- 4 프로그램의 계산 복잡도 (program complexity)
- 5 타입으로 정리하기 (types & typeful programming)
- 6 맞는 프로그램인지 확인하기 (program correctness)

프로그램 구성에 필요한 요소

- ▶ 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

(pictures from Google search)



컴퓨터 프로그램이 다른 점: 만든 것이 실행(계산)된다

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	()	

- ▶ 기본 타입들: $\mathbb{N}, \mathbb{R}, \mathbb{B}, \text{String}, \text{Symbol}$
- ▶ 실행(evaluation, semantics): -1.2e2는 -1.2×10^2 , #t는 참, +는 +, 'snow는 "snow" 라는 심볼, 등

식을 조합하는 방법

$P ::= E$	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):
 - ▶ 만들 수 있는 식은 무한히 많음
 - ▶ 식안에 임의의 식들을 맘껏 조합할 수 있음
- ▶ 조합식의 실행(semantics)은 어떻게 될까? 그 실행을 머릿속에 그려야.

프로그램 식의 실행과정

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

주의:

- ▶ 식의 실행 규칙(rule of evaluation, semantics): 명확히 정의됨
- ▶ 프로그래머는 이것을 이해해야 의도한 프로그램을 작성할 수 있음
- ▶ 제대로 실행될 수 없는(오류있는) 멸절된 식들이 많음

식의 실행규칙(rule of evaluation, semantics)

실행 규칙. 각 식의 종류에 따라서:

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

주의:

- ▶ 생긴게 옳다고 모두 제대로 실행되는 게 아님
- ▶ 부품식들의 계산결과와 타입이 맞아야
- ▶ 이름의 사용, 이름의 유효범위(scope)

프로그램식 조합방식의 원리



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



이 원리를 확인해보면

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

다음

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

이름짓기(binding, declaration, definition)

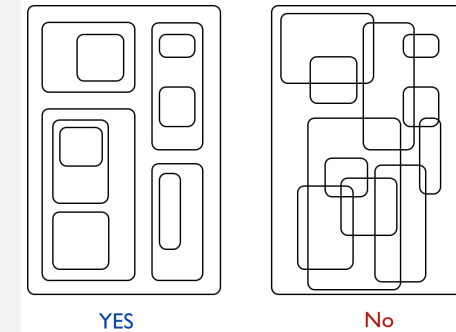


이름 짓기는 속내용감추기(abstraction)의 첫 스텝: 이름을 지으면 지칭하는 대상(속내용) 대신에 그 이름을 사용

- ▶ 이름 지을 수 있는 대상:
 - ▶ 프로그램에서 다룰 수 있는 모든 값
- ▶ 이름의 유효범위(scope)가 한정됨. 따라서,
 - ▶ 이름 재사용 가능
 - ▶ 전체 프로그램의 모든 이름을 외울 필요없슴
 - ▶ 이름이 필요한 곳에만 알려짐
- ▶ 이름의 유효범위(scope)는 쉽게 결정됨

이름의 유효범위(scope) 결정

프로그램 텍스트에서 쉽게 결정됨(lexical scoping)



이런 간단한 유효범위는 수리논술의 2000년 전통:

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

이름짓기(binding, declaration, definition)

- ▶ 식에서 이름짓기

$E ::= \dots$	예전것들
$(\text{let } ((x E)^+) E)$	x 의 정의
$(\text{letrec } ((x E)^+) E)$	x 의 재귀정의

- ▶ 프로그램에서 이름짓기

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

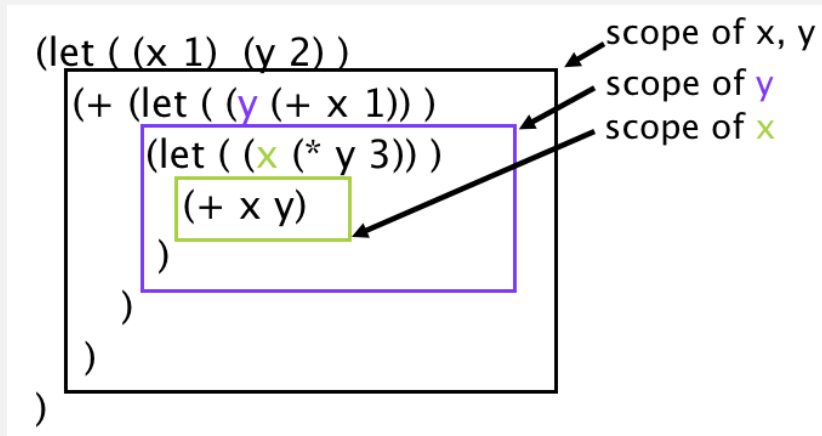
이름짓기의 실행규칙(rule of evaluation, semantics)

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

주의:

- ▶ 생긴게 옳다고 모두 제대로 실행되는 게 아님
- ▶ 부품식들의 계산결과의 타입이 맞아야
- ▶ 이름의 유효범위(scope)
- ▶ 환경(environment): 이름과 그 대상(값)의 목록표

이름의 유효범위(scope) 예



이름짓기 + 사용하는의 실행과정(semantics)

컴퓨터는 프로그램 식을 실행할 때

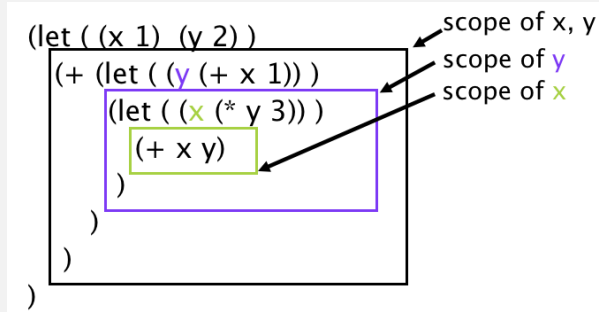
- ▶ 이름과 그 대상의 목록표를 관리
- ▶ 그러한 목록표를 환경(environment)이라고 함

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

이름짓기 + 사용하는의 실행과정(semantics)

환경(environment) 관리

- ▶ 환경 만들기: 이름이 지어지면
- ▶ 환경 참조하기: 이름이 나타나면
- ▶ 환경 폐기하기: 유효범위가 끝나면



여러개 한꺼번에 이름짓기: 실행의미

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

설탕구조(syntactic sugar)

편리를 위해서 제공; 지금까지것들로 구성가능; 반드시 필요는 없다:

list, cond, let, define은 설탕

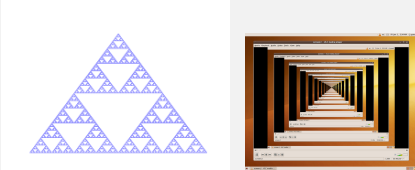
```
(list E*)           = (cons ...)  
(cond (E E') (else E'')) = (if ...)  
(let ((x E)) E')    = ((lambda ...)  
(let ((x1 E1) (x2 E2)) E) = ((lambda ...)  
(define x E) E'     = (letrec ...)  
(define x E) (define y E') E'' = (letrec ...)  
(define (f x) E)    = (define ...)  
(begin E E')       = ((lambda ...)
```

다음

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

재귀(recursion): 되돌기, 같은 일의 반복

▶ 예) 재귀하고있는 그림들:



▶ 예) 재귀하고있는 표기법:

```
E ::= c  
    | x  
    | (if E E E)  
    | (cons E E)
```

▶ 예) 재귀하고있는 정의:

```
a0 = 1, an+1 = an + 2 (n ∈ ℕ)  
X = 1 ↔ X
```

재귀함수(recursive function)의 정의

▶ 함수만 재귀적으로 정의가능 (대부분의 언어) (왜?)

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

▶ 임의의 값을 재귀적으로 정의? 그 값 계산이 무한할 수 있음

```
(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 자동 변환
 - ▶ 끝재귀(tail recursion) 변환

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))
```

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

$$\int_{n=a}^b f(n) = f(a) + \dots + f(b)$$

- ▶ 함수가 결과로

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

현대 프로그래밍에서

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

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 upper)
  ))

(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)
```

다음

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

계산 복잡도(complexity)

프로그램 실행 비용의 증가정도(order of growth)

- ▶ 계산비용 = 시간과 메모리
- ▶ 증가정도 = 입력 크기에 대한 함수로, 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:

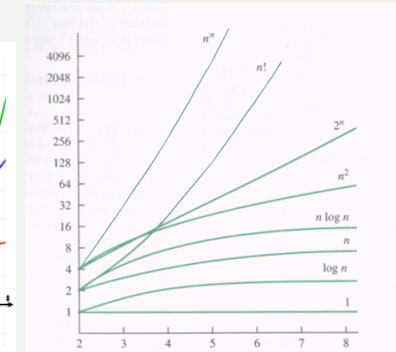
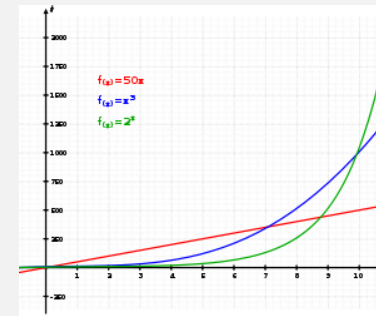
$$k_1 \times f(n) \leq \bullet \leq k_2 \times f(n)$$

(k_1, k_2 는 n 과 무관한 양의 상수)

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

계산복잡도(complexity)

(pictures from Google search)



계산복잡도(complexity)

- ▶ (fac n): 시간복잡도 $\Theta(n)$, 메모리복잡도 $\Theta(n)$.
- ▶ (exp b n):
 - ▶ $\Theta(n)$ 로 구현가능
 - ▶ $\Theta(\log n)$ 로 구현가능
- ▶ (sat formula):
 - ▶ $\Theta(2^n)$ 로 구현가능
 - ▶ $\Theta(\text{poly}(n))$ 로 구현가능? 누구도모름
- ▶ (diophantine eqn):
 - ▶ $\Theta(2^n)$ 로 구현가능? 누구도모름
 - ▶ $\Theta(n^n)$ 로 구현가능? 누구도모름

다음

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

타입(type)



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

- ▶ 타입(type)은 가이드다
 - ▶ 프로그램의 실행안전성을 확인하는
 - ▶ 새로운 종류의 데이터값을 구성하는

사용하는 타입들(types)

$\tau ::= \iota$	primitive type
$\tau \times \tau$	pair(product) type
$\tau + \tau$	or(sum) type
$\tau \rightarrow \tau$	fn type, single param
$\tau * \dots * \tau \rightarrow \tau$	fn 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

고차함수 타입 예:

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

타입을 상상하며 식을 구성하기

- ▶ $c : \iota$
- ▶ $x : \tau$
- ▶ $(\text{if } E:\mathbb{B} \ E':\tau \ E'':\tau) : \tau$
- ▶ $(\text{lambda } (x:\tau) \ E:\tau') : \tau \rightarrow \tau'$
 - ▶ $x : \tau$ 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

- ▶ $c : l$
- ▶ $x : \tau$
- ▶ $(\text{if } E:\mathbb{B} \ E':\tau \ E'':\tau) : \tau$
- ▶ $(\text{lambda } (x:\tau) \ E:\tau') : \tau \rightarrow \tau'$
 - ▶ $x : \tau$ 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

- ▶ $(\text{let } ((x_1:\tau_1 \ E_1:\tau_1) \ (x_2:\tau_2 \ E_2:\tau_2)) \ E:\tau)$
 - ▶ $x_1 : \tau_1$ 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

타입으로 프로그램을 정리/검산하기

```
(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))))
))
```

타입으로 프로그램을 정리/검산하기

```
(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 프로그램의 계산 복잡도 (program complexity)
- 5 타입으로 정리하기 (types & typeful programming)
- 6 맞는 프로그램인지 확인하기 (program correctness)

맞는 프로그램인지 확인하기

프로그램을 돌리기 전에 (static test)

- ▶ 분석 검증 후 프로그램 제출/출시/탑재
- ▶ 다른 공학분야와 동일:
 - ▶ 기계/전기/공정/건축설계 분석 검증 후 제작/설비/건설
- ▶ 일상과 동일:
 - ▶ 입시/면접, 사주/궁합, 클럽기도
 - ▶ 검증 후 실행

4190.210에서는 간단한 기술만

검증해야 할 성질들

- ▶ 제대로 생겼는가? 자동검증
- ▶ 타입에 맞게 실행될 것인가?
 - ▶ 직접검증(Scheme, C, JavaScript, etc)
 - ▶ 자동검증(ML, Scala, Haskell, Java, Python, etc)
- ▶ 4190.210: 모든 입력에 대해서 정의되었는가?
직접검증, 용이
- ▶ 4190.210: 항상 끝나는가: 직접검증, 비교적 용이
- ▶ 내가 바라는 계산을 하는가: 어려움

확인: 모든 입력에 대해서 정의되었는가?

- ▶ 타입에 맞게 실행될 것이 확인된 경우
- ▶ 타입에 맞게 실행될 것이 확인안된 경우
데이터 구현을 익히고 나서.

확인: 항상 끝나는가?

그렇다, 만일:

- ▶ 반복 될 때 뭔가가 계속 “줄어들고”
- ▶ 그 줄어듬의 “끝이 있다”면.

즉, 재귀함수의 경우, 만일:

- ▶ 재귀 호출마다 인자가 “줄어들고”
- ▶ 그 줄어듬의 “끝이 있다”면.

끝나는 재귀함수인지 확인하기

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

- ▶ 음이 아닌 정수만 입력으로 받는다면,
- ▶ 재귀호출마다 원래 n 보다 줄고 있고 “ $n-1$ ”,
- ▶ 끝이 있다(“(= n 0) 1”).

끝나는 재귀함수인지 확인하기

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

일반방법: 끝나는 재귀함수인지 확인하기

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

- ▶ 원소들 간의 줄어드는 순서 $>$ 를 찾으라, 아래와 같은:
- ▶ 재귀호출 인자식(e_1 와 e_2)의 값이 원래 인자(x)보다 $>$ 인(“줄어드는”), 그리고
- ▶ 집합 A 에서 $>$ -순서대로 원소를 줄세우면 항상 유한한(finitely well-founded).

끝나는 재귀함수인지 확인하기

```
(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?
- ▶ 그래서 그 $>$ -순서가 항상 유한번에 바닥에 닿는(finitely well-founded)?

그런 순서 $>$ 는?

끝나는 재귀함수인지 확인하기

```
(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 ()))
```

끝나는 재귀함수인지 확인하기

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

끝나는 재귀함수인지 확인하기

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

끝나는 재귀함수인지 확인하기

$$f(\epsilon, c) = \emptyset$$

$$f(c', c) = \emptyset \quad (c \neq c')$$

$$f(c, c) = \{\epsilon\}$$

$$f(r_1|r_2, c) = f(r_1, c) \cup f(r_2, c)$$

$$f(r^b, c) = \{r'r^b \mid r' \in f(r, c)\}$$

$$f(cr_2, c) = \{r_2\}$$

$$f(c'r_2, c) = \emptyset \quad (c \neq c')$$

$$f(\epsilon r_2, c) = f(r_2, c)$$

$$f((r_{1_1}r_{1_2})r_2, c) = f(r_{1_1}(r_{1_2}r_2), c)$$

$$f((r_{1_1}|r_{1_2})r_2, c) = f(r_{1_1}r_2, c) \cup f(r_{1_2}r_2, c)$$

$$f(r_1^b r_2, c) = f(r_2, c) \cup \{r'r_1^b r_2 \mid r' \in f(r_1, c)\}$$