— Functional programming

Computation: mathematical function mapping inputs to outputs.

→ no need for assignment statement

pure functional paradigm — no var, no assign st., no loop (→ recursive way)

ex) loop → recursive way

(X^Y)

```
int power (int x, int y)
{ result = x;
  for (int i=1 ; i<y; i++)
      result = result * x ;
  return result;
}
```

∃ loop, assign st.

⇒

```
int power(int x, int y)
{ if (y==0)
      return 1 ;
  else
      return (x * power(x, y-1));
}
```

∄ loop, assign st.

— pure func. prog. Lang.

provides better supports for program correctness proofs. (than imperative lang.)

∵ ∄ loop, ∄ assign statement — needs complex proof rules for correctness

(assign statement: changes system state)

uses functions and linear data structure (list)

∄ concept of memory cell (var), update memory (assign st.)

Function

ex) $\underbrace{square}_{\text{func. name}}(\underbrace{n}_{\text{arg.}}) = \underbrace{n * n}_{\text{expression}}$

func. name arg. expression (defines the meaning of the func.)

$square : \underbrace{\overset{\text{real\#}}{R}}_{\text{domain}} \to \underbrace{\overset{\text{real\#}}{R}}_{\text{range}}$

- total func. — defined for all elements in its domain (input)
- partial func.

— square is a total func. (defined for all real#)
domain ≡ input values

Values of an expression depends only on the values of subexpressions (if any).

ex) value of $a+b$ — depends on values of a and b

∄ assignment statement

∄ previous/next state.

Lambda Calculus

lambda expression — Specifies parameter and func. definition, but ∄ func. name.

ex) $(\lambda \underbrace{x}_{\text{para. (id)}} \cdot \underbrace{x * x}_{\text{body}})$ — a func. definition without func. name.

$((\lambda x \cdot x * x)\ 2))$ — func. application

$\Rightarrow 4$

— pure Lambda Calculus — by Church (1941)

1. any id is a lambda expr.
2. if M and N are lambda expr's,
   then, application of M to N $(M \underset{\text{arg}}{N})$ is a lambda expr.
3. An abstraction $(\lambda \underset{\text{id}}{x} \cdot \underset{\text{Lambda expr.}}{M})$ is also a lambda expr.

ex)
lambda expr's
x
$(\lambda x \cdot x)$
$((\lambda x \cdot x)\ (\lambda y \cdot y))$

— meaning of the lambda expr. is defined by
β-reduction rule

$$\boxed{((\lambda \underset{\text{para}}{x} \cdot \underset{\text{func. body}}{M})\ \underset{\text{arg}}{N}) \underset{\text{β-reduction}}{\Rightarrow} M[N/x]}$$

in M (func. body),
all x's are substituted by N arg.

— evaluation of a lambda expression
(= a sequence of β-reductions)

ex) $((\lambda y \cdot ((\lambda x \cdot xyz)\ a))\ b)$

inner-most eval
$\Rightarrow ((\lambda y \cdot ayz)\ b)$
$\Rightarrow abz$

outer-most eval
$\Rightarrow ((\lambda x \cdot xbz)\ a)$
$\Rightarrow abz$

Same result (must be)

graphical
↓

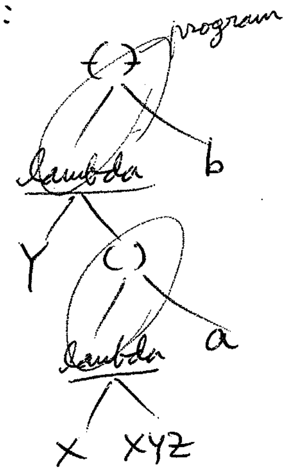
ex) graphical notation (β-reduction)

program program

$\{((\lambda Y \cdot ((\lambda x \cdot xyz)\ a))\ b\}$

arg. arg.

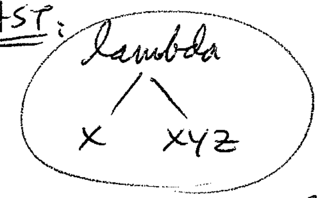
AST:



ref para body

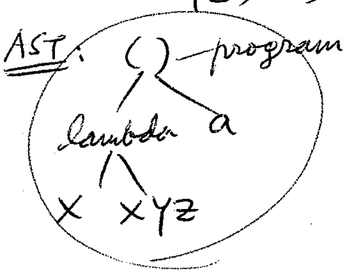
— $(\lambda x \cdot xyz)$

AST:



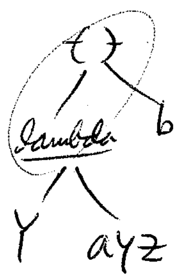
— $((\lambda x \cdot xyz)\ a)$ arg

AST:



inner-most reduction

outer-most reduction



$\Downarrow$

abz ✓

$\Downarrow$

abz ✓

same

— func. prog. Lang's to study

- LISP — untyped, dynamic scope, prefix expr, ∄ assignst, List for prog/data.
- Scheme — untyped, static scope, prefix expr, ∄ assignst, "
- ML — strongly typed, static scope, infix expr., ∃ assignst, X

ex) (let val $x=3$
in x^2-x+1

— Scheme syntax

ex) Expression → value

(+ 2 3) ⟹ 5

'(+ 2 3) ⟹ (+ 2 3)

(car (cons '1 '(2 3))) ⟹ 1

ex) (cons '1 '(2 3)) ⟹ (1 2 3)
(car '(1 2 3)) ⟹ 1
(cdr '(1 2 3)) ⟹ (2 3)

'1 ⟹ 1

'(2 3) ⟹ (2 3)

(2 3) ⟹ error

treat as a func. (prefix)

(car (cons 1 (2 3))) ⟹ error

func X

'(car (cons 1 (2 3))) ⟹ (car (cons 1 (2 3)))

(car '(cons 1 (2 3))) ⟹ cons

(list (+ 2 3) 7) ⟹ (5 7)

(has any # of args

(quote ((+ 2 3) 7)) ⟹ ((+ 2 3) 7)

1

(has only 1 arg.

((+2 3) 7) ⟹ error

Scheme

– function definition

ex) (define (square x) (* x x))

$\underbrace{\text{square}}_{\text{func. name}}$ $\underbrace{x}_{\text{para}}$ $\underbrace{(* \; x \; x)}_{\text{body}}$

(square 5) ⇒ 25

$\underbrace{\text{(square 5)}}_{\text{func. application}}$

or, using lambda func.

(define square (lambda (x) (* x x)))

(square 5) ⇒ 25

meaning

square ≡ (lambda (x) (* x x))

– map – a built-in func.

ex) (define (f x) (+ 1 x))

f: func. name, x: para

(map f '(1 2 3)) ⇒ (2 3 4)

$\underbrace{\text{(map f '(1 2 3))}}_{\text{apply func. f on a list of values.}}$

ex) (define (inc x) (+ 1 x))

(inc 3) ⇒ 4

(map inc '(1 2 3 4)) ⇒ (2 3 4 5)

ex) (map (lambda (x) (+ 1 x)) '(1 2 3 4)) ⇒ (2 3 4 5)

— Running Scheme interpreter

— Interactive way:

```
$> scheme ↵
1]=> (* 5 8)
; value: 40
  ⋮
1]=> Crl-C ↵
interrupt option (? for help): Q ↵
$>
```

or, Crl-D. to quit the session.

ex)
```
1]=> (define (inc x)
      (+ x 1)
     )
; value: inc
1]=> (inc 3)
; value: 4
```

— Using file (source code stored in a file):

Script →

1. create a file with a source code and save;
2. invoke the scheme interpreter: $> scheme ↵
3. Load the programe file: 1]=> (load "file-name") ↵
4. type expressions to be evaluated: 1]=> (fibo 5)
5. quit scheme session: 1]=> Crl-D ↵
   or 1]=> Crl-C ↵ and Q ↵

exit →

for typescript file

fibo: a func. name defined in the prog.

5: arg.

ex)
```
(define (fibo x) (---))
  ⋮
(define (inc x) (----))
  ⋮
```

— Conditionals (if-else like)

Syntax

```
(if P E1 E2)
```

Semantic

if P → E_1
else → E_2

ex)

```
(define (fact n)
  (if (= n 0)            ; P
      1                  ; E1
      (* n (fact (- n 1)))  ; E2
  )
)
```

— Branch

Syntax

```
(cond (p1 E1)
      (p2 E2)
        ⋮
      (else En)
)
```

Semantic

if P_1 → E_1
elsif P_2 → E_2
⋮
else → E_n

ex)

```
(define (fact n)
  (cond ((= n 0) 1)          ; P1, E1
        (else (* n (fact (- n 1))))   ; En
  )
)
```

0! = 1

— Let construct

Assume:

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

or

```
(define (square x)
  (* x x)
)
```

ex)

```
(let ((three-sq (square 3))
      (four-sq (square 4))
     )
  (+ three-sq four-sq)
)      ; three-sq = 9, four-sq = 16
```

⇒ 25

scheme syntax

— Structure of List (LISP/Scheme)

both program/data are lists

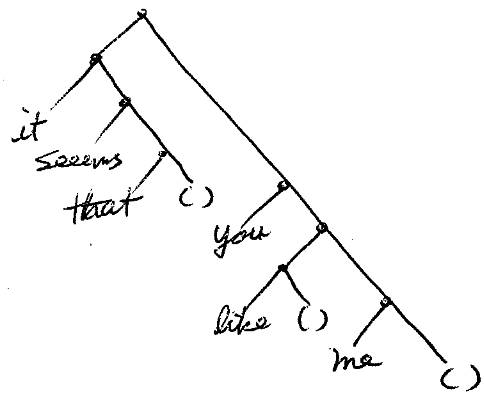
elements of list canbe

booleans, numbers, symbols, lists, functions, ---

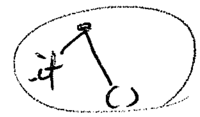
e.g) ((it seems that) you (like) me) — 4 ele. list

(it seems that) — 3 ele. list

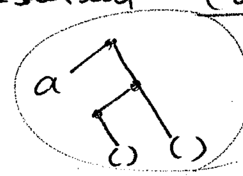
(like) — 1 ele. list



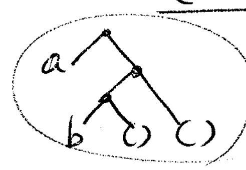
e.g) 1 ele. list — (it)



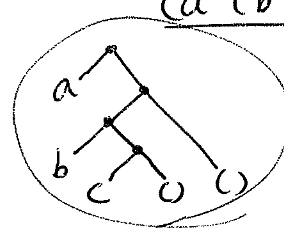
2 ele. list — (a ())



(a (b))



(a (b c))



— operations on list

build lists (in Scheme syntax)

(define x '((a b c) d (e) f))

(define g '(1 2))

(null? x) ⇒ #f

(car x) ⟶ (a b c)

(cdr x) ⟶ (d (e) f)

(cons g x) → ((1 2) (abc) d (e) f)

(car (cons g x)) → (1 2)

(cdr (cons g x)) → ((abc) d (e) f)

(car (car x)) ≡ (caar x) ⟶ a

(cdr (car x)) ≡ (cdar x) → (b c)

(car (cdr x)) ≡ (cadr x) → d

(cdr (cdr x)) ≡ (cddr x) → ((e) f)

— List manipulation

Cons — builds list

ex) Create 3 element list (a b c)

```
(cons 'a (cons 'b (cons 'c '()))) ⇒ (a b c)
```

(c)

(b c)

(a b c)

or,

```
(list 'a 'b 'c) ⇒ (a b c)
```

— in dot (·) notation

```
'(a · (b · (c · ()))) ⇒ (a b c)
```

— Cons/append

ex)

```
(cons '(a b c) '(d)) ⇒ ((a b c) d)
(append '(a b c) '(d)) ⇒ (a b c d)
```

ex)

```
(cons 'a (append '(b c) '(d))) ⇒ (a b c d)
```

— length — built-in func.

```
(length '()) ⇒ 0
(length '(1 2)) ⇒ 2
```

```
(define (length x)
  (cond ((null? x) 0)
        (else (+ 1 (length (cdr x))))
  )
)
```

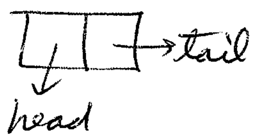
— Storage allocation/deallocation for Lists (in LISP dialects)

(garbage collection)

— memory allocation by cons
alloc. single cell

Cons returns a pointer (addr.) to a newly allocated cell.

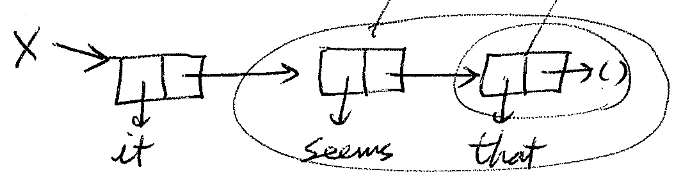
a cell



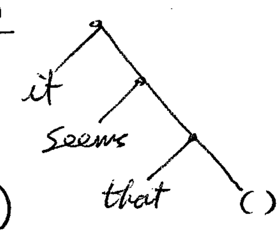
(car — returns pointer in the 1st field
cdr — returns pointer in the 2nd field.

in Scheme syntax,

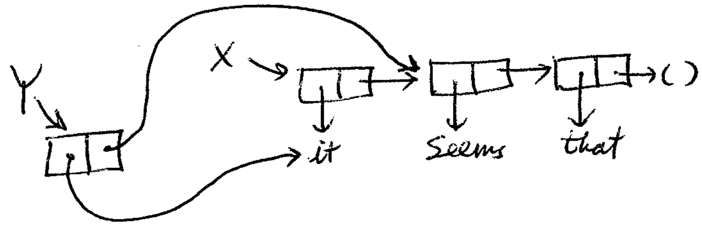
ex) (define x (cons 'it (cons 'seems (cons 'that '()))))



⇒ (it seems that)



(define Y (cons (car x) (cdr x)))



⇒ (it seems that)
list Y

(check X, Y holds same elements

(equal? x Y) ⇒ #t

(eq? x Y) ⇒ #f

check x and Y point same cell.

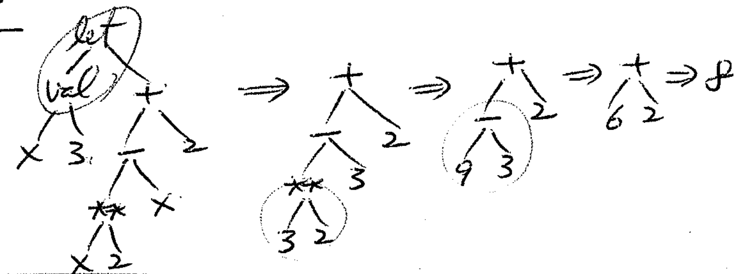
— ML — Lexical (static) scope
Strongly typed, infix expression

↓
ML

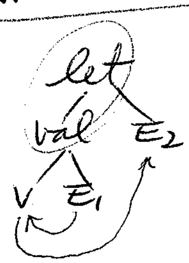
let expression evaluation

ex) let val X=3 in
    X**2 − X + 2
end

AST



(reduction)
— Rewrite rule for let expression



$\Rightarrow E_2[V/E_1]$

V in E_2 is replaced by E_1

ex) let val x=3 in
    x**2 − x + 2
end

$\Rightarrow \underbrace{(x**2-x+2)}_{E_2}\,[\underbrace{x}_{V}/\underbrace{3}_{E_1}]$

$\Rightarrow 3**2-3+2$

$\Rightarrow 9-3+2$

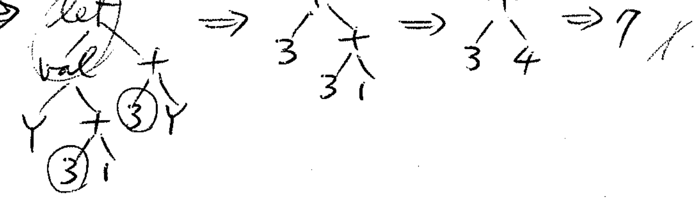
$\Rightarrow 6+2$

$\Rightarrow 8$

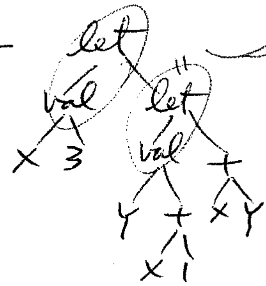
---

ex) let val x=3 in
    let val Y = X+1 in
        X+Y
    end
end

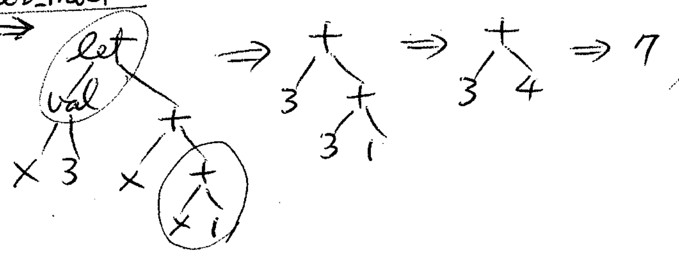
outer-most reduction



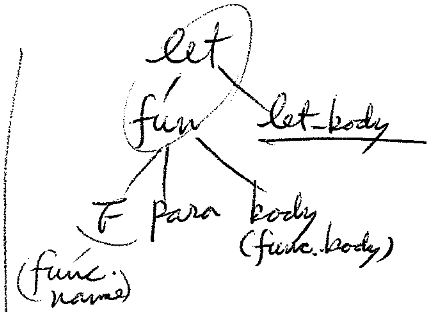
AST



inner_most



— Rewrite rule for functions



⇒ Let-body [F(arg) / Func.body [para/arg]]

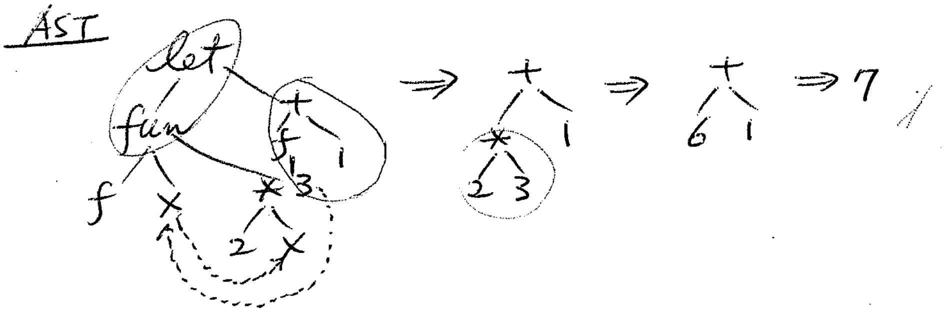
(in Let-body, replace all func.call (with arg) with func.body in which para ← arg.

ex) let fun f(x) = 2 * x in
f(3) + 1
end

para; func.body; arg; let body

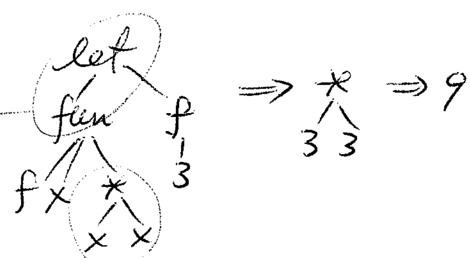
ref. $E_2 [V/E_1]$
⇒ in E_2, replace all V with E_1

AST

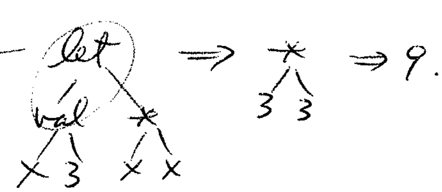


---

ex) let fun f(x) = x * x in
f(3)
end



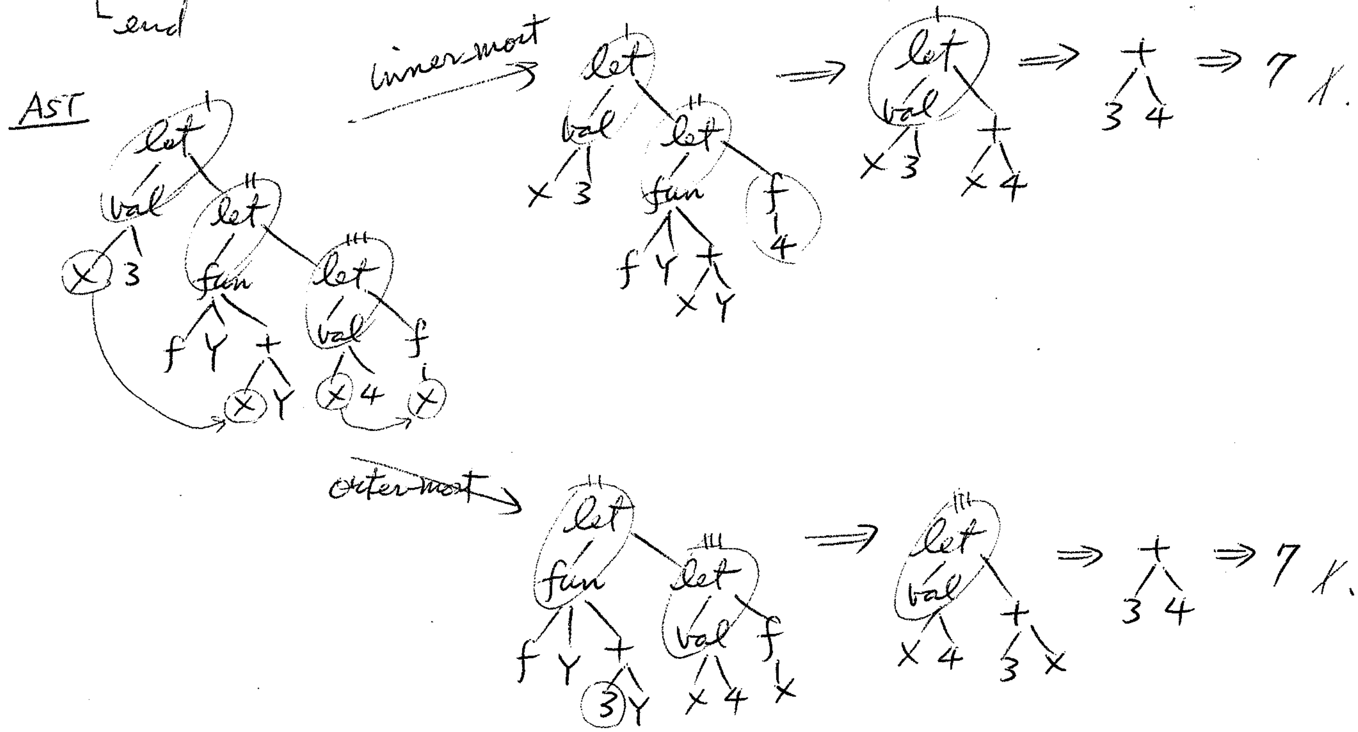
≡ let val x=3 in
x * x
end



HW

— Show both innermost/outermost reductions for

```
let val x=3 in
    let fun f(y)=x+y in
        let val x=4 in
            f (x)
        end
    end
end
```



— nested binding

$$\left(\begin{array}{l}\text{let val } x_1 = E_1 \\ \quad \text{val } x_2 = E_2 \text{ in} \\ \quad E \\ \text{end}\end{array}\right) \Rightarrow \begin{array}{l}\text{let val } x_1 = E_1 \text{ in} \\ \quad \text{let val } x_2 = E_2 \text{ in} \\ \quad\quad E \\ \quad \text{end} \\ \text{end}\end{array}$$

— Simultaneous binding

$$\begin{array}{l}\text{let fun } f_1(x_1) = E_1 \\ \text{and } f_2(x_2) = E_2 \text{ in} \\ \quad E \\ \text{end}\end{array} \longrightarrow \begin{array}{l}\text{Scope of } x_1 \text{ is } E_1 \\ \text{Scope of } x_2 \text{ is } E_2 \\ \text{Scope of } f_1 \text{ and } f_2 \text{ includes} \\ \quad E_1, E_2, E.\end{array}$$

→ 4) let fun even(x) =

(E_1) if $x=\phi$ then T else if $x=1$ then F else odd$(x-1)$

and odd(x) =

(E_2) if $x=\phi$ then F else if $x=1$ then T else even$(x-1)$

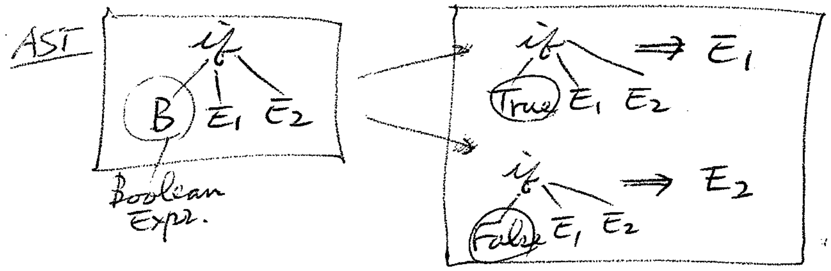
(E) in (even(4), odd(4))

end

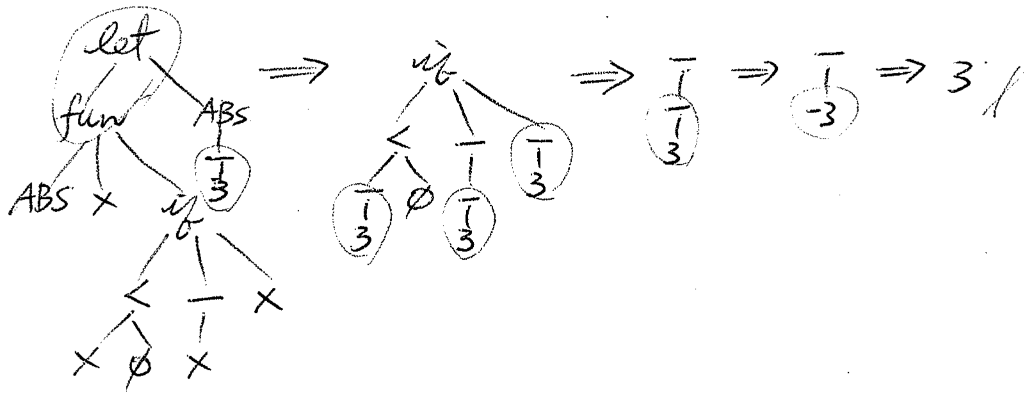
$\Rightarrow$ (T, F)

— Scope of both even and odd includes E1, E2, and E.

— Rewrite rule for Conditional expression



2)
```
let fun ABS(x) =
      if x < 0 then -x
               else x
   in ABS(-3)
end
```



in ML,

- overloading (multiple meaning)

  예) +, * are overloaded operations — ML supports

  2+2 → int
  int int

  2.0 + 2.0 → real
  real real

- Coercion (implicit type conversion)

  ML dose not use coercion.

  uses explicit type conversion (type casting)

  예) 2*3.4 ⇒ error.

  real(2) * 3.4 ⇒ real (ok).

- polymorphism : parameterized types — ML supports.

  예) hd [1, 2, 3] ⇒ 1 — (int list ⇒ int)

  hd ["a", "b", "c"] ⇒ "a" — (string list → string)

  hd is a polymorphic function

  func. type depends on argument type.

ML

L