Lisp
(Functional Programming Language)

국민대학교 컴퓨터공학부
강 승 식

Lisp(list processing language)

- Functional programming language
  - declarative(not procedural) language

  - LISP, Scheme, ML, Haskell
  - common Lisp: hybrid language

  - control structure: tail recursion
  - data type: single linked list
  - evaluation of a list

  - λ-calculus : Church(1940's)
  - interpreter language(compiler?)

λ-calculus : generalized function model

양 식	함수정의	실인자
$(\lambda x.\lambda y.x+y)$ 2 3	$\lambda x.\lambda y.x+y$	2 3
$(\lambda x.x)\lambda y.y$	$\lambda x.x$	$\lambda y.y$

예) $(\lambda x.\lambda y.x+y)2,3 = (\lambda y.2+y)3 = 2+3 = 5$

$(\lambda x.xy)(\lambda z.z+1) = (\lambda z.z+1)y = y+1$

λ-calculus vs. LISP

- Function definition:

  (lambda (x y) (+ x y)) → (defun add(x y) (+ x y))

  // 'defun' is a macro

- Parameter application:

  ((lambda (x y) (+ x y)) 2 3)

Data Type: symbol

- *Atomic Symbol : atom (or symbol)*
  - Numeric atom, non-numeric atom
    - numeric atom: number
    - non-numeric atom: symbol

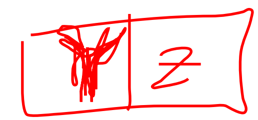
- Basic data type: S-expression

Data Type: S-expression

- S-expression : unique data type in LISP

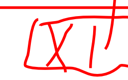
  <S-expression> ::= <atomic symbol>

  | (<S-expression> . <S-expression>)

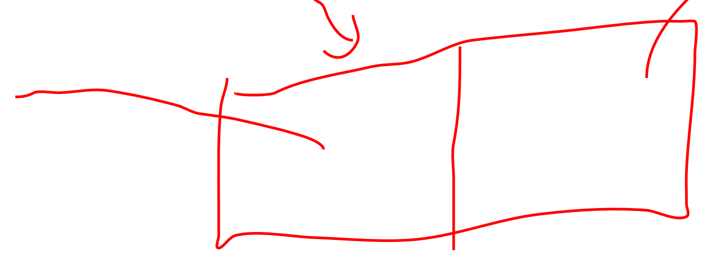


- 예) ABC, (A . B), ((A . B) . C), ((A . B) . (X . (Y . Z)))

(a.(b.(c.() < - nil)))



- S-expression is implemented by binary tree



Data Type: list

- list

  (A B C D) or (A, B, C, D)

  → (A . (B . (C . (D . NIL))))

- NIL
  - () -- empty list
  - both a list and an atom

(A (B C) D) → (A . ((B . (C . NIL)) . (D . NIL)))

Primitive LISP functions

- All functions have a return value.

- List manipulation functions

(cons ‘x ‘y) → (x . y)
(car '(x . y)) → x
(car '(x y z)) → x
(cdr '(x . y)) → y
(cdr '(x y z)) → (y z)

(setq x '(a b c)) → x = (a b c)
(list ‘x ‘y) → (x y)

Arithmetic calculation

(+ 1 2 3) → 6

(- 5 3) → 2

(* 2 3 4) → 24

(/ 12 3) → 4

(1+ 2) → 3

(1- 3) → 2

Comparison

- Return value is: t (true) or nil (false)

  (atom 'a) → t

  (atom 1.5) → t

  (atom '(a b)) → nil

  (atom nil) → ?

  (null nil) → t

  (null ())→ t

  (null 'x) → nil

  (null '(a b)) → nil

  (eq 'x 'x) → t

  (eq nil ())→ t

  (eq (car '(x y)) 'x) → t

※ eq: atom comparison,
equal: compare S-expressions

※ Relational expression -- AND, OR, NOT

Data or Statement: Quote

- (quote a)
- ‘a

- (set (quote a) ‘(x y z))
- (set ‘a ‘(x y z))
- (setq a ‘(x y z))

- (car a) vs. (car ‘a)
- (cdr a) vs. (cdr ‘a)

Evaluation

- (setq a ‘(cdr ‘(x y z)))
- (eval a)

- Data and statement has no difference.
- They are all represented by a list (S-expression).

Control structure

$$
\begin{array}{l}
\text{(cond (test}_1\ \text{result}_1\text{)} \\
\qquad\quad \text{(test}_2\ \text{result}_2\text{)} \\
\qquad\qquad \ldots \\
\qquad\quad \text{(test}_{n-1}\ \text{result}_{n-1}\text{)} \\
\qquad\quad \text{(t result}_n\text{))}
\end{array}
$$

$$
\begin{array}{l}
\text{(do ((var}_1\ \text{init}_1\ \text{step}_1\text{)} \\
\qquad\quad \text{(var}_2\ \text{init}_2\ \text{step}_2\text{)} \\
\qquad\qquad\quad \ldots\ \text{)} \\
\qquad \text{(end-test . result)} \\
\qquad \text{body} \quad \text{// LISP statements} \\
\text{)}
\end{array}
$$

User-defined functions

```
(defun func_name (v1 v2 ...... vn)
        (lisp statement)
)
```

- Example

```
(defun average (x y) (/ (+ x y) 2))

(defun fact (n)
        (cond ( (<= n 1) 1 )
              (  t  (* n (fact (1- n))) )))
```

```
(defun append (x y)
        (cond ( (null (cdr x)) (cons (car x) y) )
              (  t  (cons (car x) (append (cdr x) y)) )))

(defun reverse (x)
        (cond ( (null (cdr x)) x )
              (  t  (append (reverse (cdr x)) (list (car x))) )))
```

※ Procedural programming

prog, progn → hybrid language

Example of LISP program

```
% lisp
  > (load “filename.l”)
  > (fact 3)
    6
  > (setq x (append '(a b c) '(d e)) )
  > (reverse x)
```

※ symbol vs. string

String is a “character sequence”, or a symbol ?

※ Tail-recursion

※ A.I. programming

: symbolic processing

: declarative rather than procedural

: good writability for A.I. applications

※ Disadvantages of LISP: efficiency

- Out of memory: garbage collection
- LISP machine → symbolic

Lisp programming practice

1.1 last-item

(last-item '(a b c d)) → d

(last-item '(1 2 (3 4))) → (3 4)

(last-item '(a)) → a

(last-item '((a))) → (a)

1.2 remove-1st

(remove-1st 'a '(b a n a n a)) → (b n a n a)

(remove-1st 'a '(b (a n) a n a)) → (b (a n) n a)

(remove-1st '(1 2) '(1 2 (1 2) 3)) → (1 2 3)

(remove-1st 'cat '(dog fox hen)) → (dog fox hen)

(remove-last '(1 2) '(1 2 (1 2) 3)) → (1 2 3)

1.3 append

```
(append '(a b) '(c d e))        → (a b c d e)
(append '() '(a b c))           → (a b c)
(append '(a b c) '())           → (a b c)
(append nil nil)                → nil
```

1.4 merge

```
(merge '(1 3 4 7) '(2 4 5))     → (1 2 3 4 4 5 7)
```

1.5 subst-all

```
(subst-all 'dog 'cat '(my cat is cute)) → (my dog is cute)
(subst-all 'b 'a '(c a b a c))          → (c b b b c)
(subst-all '(0) '(*) '((*) (*) (0)))    → ((0) (0) (0))
```

2.1 count-all

(count-all '((a b) c () ((d (e))))) → 6

(count-all '(() ())) → 2

(count-all '((()))) → 1

(count-all '()) → 0

2.2 remove-all

(remove-all 'a '(b (a n) a n a)) → (b (n) n)

(remove-all 'a '((a b (c a)) (b (a c) a))) → ((b (c)) (b (c)))

(remove-all '(a b) '(a (a b) ((c (a b)) b))) → (a ((c) b))

2.3 reverse-all

(reverse-all '((1 2) (3 4) 5)) → (5 (4 3) (2 1))

(reverse-all '(a (b c) (d (e f)))) → (((f e) d) (c b) a)

2.4 depth

(depth '()) → 0

(depth '(a b c)) → 1

(depth '(a (b c))) → 2

(depth '((a b) c (d ((f))))) → 4

2.5 flatten

(flatten '((a b) c (d ((f))))) → (a b c d f)

(flatten '((a b (r)) a c (a d ((a (b)) r) a))) → (a b r a c a d a b r a)

3.1 memberp

(memberp 'b '(a b c)) → T

(memberp 'k '(l m n)) → NIL

(memberp '(e) '((c)(d)(e)(f))) → T

(memeberp '() '(l e m o n)) → NIL

3.2 union

(union '(a b) '(b c d)) → (a b c d)

(union '() '(c d)) → (c d)

(union '(l m) '(n o p)) → (l m n o p)

3.3 set-diff

(set-diff '(a b c d) '(a d)) → (b c)

(set-diff '(j k l) '()) → (j k l)

(set-diff '() '(c d)) → nil

3.4 subsetp

(subsetp '(d f) '(c d e f)) → t

(subsetp '() '(j k l)) → t

(subsetp '(v e) '(q k l)) → nil

4. Infer

- $(Q_1\ Q_2\ ...\ Q_m)$와 $(Q\ (P_1\ P_2\ ...\ P_n))$의 리스트 2개를 취하여
- Q가 첫 번째 리스트에 나오지 않으면 nil을 출력하고,
- 그렇지 않으면 Q를 $P_1\ P_2\ ...\ P_n$으로 대치한 리스트 출력

```
(infer '(D E B F) '(A (B C)))    →  NIL
(infer '(D E A F) '(A (B C)))    →  (D E B C F)
(infer '(D E A F) '(A ()))       →  (D E F)
```

5. 0과 1들의 리스트들에 대하여 아래 함수들을 작성하시오.

- distance 함수 작성 -- 같은 길이의 두 개의 비트 벡터간의 차이를 계산
  - 예: (1 0 1 1)과 (0 1 0 1)의 두 벡터의 차이는 3
- closest 함수 작성 -- 비트 벡터의 리스트에서 거리가 가장 가까운 두 벡터를 찾음

```
(distance '(1 1 1 1) '(0 0 0 0))   → 4
(distance '(1 1 1 0) '(1 1 0 1))   → 2
(distance '(0 0 0 0) '(0 0 0 0))   → 0

(closest '((1 1 1 1) (0 0 0 0) (1 1 0 0) (0 0 1 1) (1 1 1 0)))
        → ((1 1 0 0) (1 1 1 0))
(closest '((0 0 1 1) (1 0 1 1) (1 0 1 1) (0 0 1 1)))
        → ((1 0 1 1) (1 0 1 1))
(closest '((1 1 0 0) (0 0 0 0) (0 0 0 1)) → ((0 0 0 0) (0 0 0 1))
```