Programming Languages 2012: Functional Programming: Lisp

(Based on [Sethi 1996] and [Steele 1990])

Yih-Kuen Tsay

1 Introduction

Interacting with a Lisp Interpreter

```
> 3.14159              ; a number evaluates to itself
3.14159
> (setq pi 3.14159)    ; bind a variable to a value
3.14159
> pi                   ; a variable evaluates to its value
3.14159
> pI                   ; pi and pI are the same name
3.14159
```

General form of a Lisp expression: $(E_1\ E_2\ \cdots\ E_k)$

```
> (* 5 7)              ; 5 * 7
35
> (+ 4 (* 5 7))        ; 4 + 5 * 7
39
```

Dialects: Scheme vs. Common Lisp

Scheme	Common Lisp
(define pi 3.14159)	(setq pi 3.14159)
(define (sq x) (* x x))	(defun sq (x) (* x x))
((lambda (x) (* x x)) 5)	((lambda (x) (* x x)) 5)
#t	t
#f	() or nil
number?	numberp
symbol?	symbolp
equal?	equal
null?	null
pair?	consp
(map sq '(1 2 3))	(mapcar (function sq) '(1 2 3)) or (mapcar #'sq '(1 2 3))
(map list '(a b c) '(1 2 3))	(mapcar #'list '(a b c) '(1 2 3))

Dialects: Scheme vs. Common Lisp (cont.)

- When f is a formal argument representing an n-ary function, the Scheme expression (f $E_1\ E_2\ \cdots\ E_n$) translates into (funcall f $E_1\ E_2\ \cdots\ E_n$) in Common Lisp.
- There is no Common Lisp counterpart of the Scheme expression (define sq (lambda (x) (* x x))).

2 A Quick Tour

Functions

```
> (defun square (x) (* x x))    ; let square x = x*x
SQUARE
> (square 5)                    ; apply function square to 5
25
```

General form of a function definition:

$$(\text{defun } \langle function-name\rangle\ (\langle formals\rangle)\ \langle expression\rangle)$$

```
> ((lambda (x) (* x x)) 5)   ; unnamed function applied to 5
25
```

General form of an unnamed function:

$$(\text{lambda } (\langle formals\rangle)\ \langle expression\rangle)$$

Conditionals

(if P E_1 E_2) ; if P then E_1 else E_2

(cond (P_1 E_1)	; if P_1 then E_1
(P_2 E_2)	; else if P_2 then E_2
...	; ...
(P_k E_k)	; else if P_k then E_k
(t E_{k+1}))	; else E_{k+1}

Example:

```
(defun fact (n)                 ; let rec fact n =
  (if (= n 0)                   ;   if n = 0
      1                         ;   then 1
      (* n (fact (- n 1))) ))   ;   else n * fact (n−1)
```

The let Construct

General form:

$$(\text{let } ((x_1\ E_1)\ (x_2\ E_2)\ \cdots\ (x_k\ E_k))\ F)$$

The let construct allows subexpressions to be named.

```
> (+ (square 3) (square 4))
25
> (let ( (three-sq (square 3))
         (four-sq (square 4)) )
       (+ three-sq four-sq) )
25
```

The let Construct (cont.)

The let construct can also be used to factor out common subexpressions.

```
> (+ (square 3) (square 3))
18
> (let ((three-sq (square 3)))
       (+ three-sq three-sq) )
18
```

The let* Construct

General form:

$$(\text{let* } ((x_1\ E_1)\ (x_2\ E_2)\ \cdots\ (x_k\ E_k))\ F)$$

The let* construct is the sequential version of let.

```
> (setq x 0)
0
> (let ((x 2) (y x)) y)   ; bind y before redefining x
0
> (let* ((x 2) (y x)) y) ; bind y after redefining x
2
```

Quoting

General form:

(quote ⟨*item*⟩) or ′⟨*item*⟩

Quoting is needed to treat expression as data.

```
> (setq pi 3.14159)
3.14159
> pi
3.14159

> (quote pi)
PI
> 'pi
PI
```

Quoting (cont.)

```
> (setq x (+ 2 3))
5
> x
5

> (setq x '(+ 2 3))
(+ 2 3)
> x
(+ 2 3)
```

Summary of Lisp Constructs

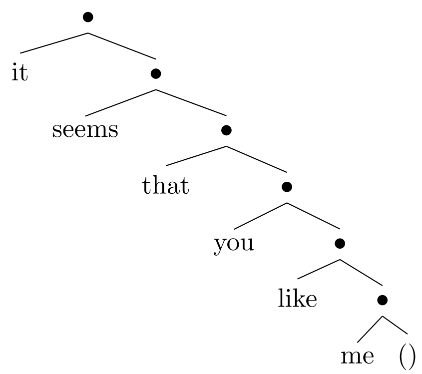
(setq pi 3.14159)	; give name pi to 3.14159
(defun sq (x) (* x x))	; fun $sq(x) = x * x$
(lambda (x) (* x x))	; anonymous function value
	; ((lambda (x) (* x x)) 3) ≡ 9
(* E_1 E_2)	; $E_1 * E_2$
(E_1 E_2 E_3)	; apply the value of E_1 as a
	; function to arguments E_2 and E_3
(if P E_1 E_2)	; if P then E_1 else E_2
(cond (P_1 E_1)	; if P_1 then E_1
(P_2 E_2)	; else if P_2 then E_2
(t E_3))	; else E_3

Summary of Lisp Constructs (cont.)

(let ((x_1 E_1)	; evaluate E_1 and E_2; then
(x_2 E_2))	; evaluate E_3 with x_1 and x_2
E_3)	; bound to their values
(let* ((x_1 E_1)	; let $x_1 = E_1$ in
(x_2 E_2))	; let $x_2 = E_2$ in
E_3)	; E_3
(quote blue)	; symbol blue
(quote (blue green red))	; list (blue green red)
(list E_1 E_2 E_3)	; list of the values of E_1, E_2, E_3

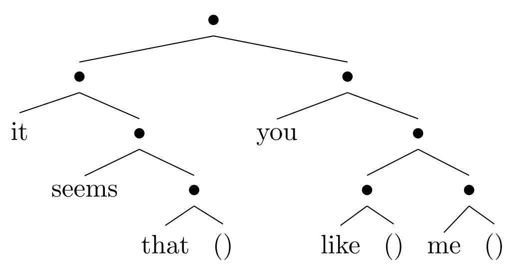
3 The Structure of Lists

Structure of a List



(it seems that you like me)

Structure of a List (cont.)



((it seems that) you (like) me)

Operations on Lists

(null x)	true if x is the empty list
(car x)	the first element of a nonempty list x
(cdr x)	the rest of the list x after the first element is removed
(cons a x)	a value with car a and cdr x; that is (car (cons a x)) ≡ a (cdr (cons a x)) ≡ x

Operations on Lists (cont.)

> (setq x '((it seems that) you (like) me)) *((IT SEEMS THAT) YOU (LIKE) ME)*

expression	shorthand	value
x	x	((it seems that) you (like) me)
(car x)	(car x)	(it seems that)
(car (car x))	(caar x)	it
(cdr (car x))	(cdar x)	(seems that)
(cdr x)	(cdr x)	(you (like) me)
(car (cdr x))	(cadr x)	you
(cdr (cdr x))	(cddr x)	((like) me)

Cons

> '(it . (seems . (that . ())))
(IT SEEMS THAT)

> (cons 'it (cons 'seems (cons 'that '())))
(IT SEEMS THAT)

> (list 'it 'seems 'that)
(IT SEEMS THAT)

4 List Manipulations

Functions on Lists

```
(defun  my-length (x)
  (cond ((null x) 0)
        (t (+ 1 (my-length (cdr x)))) ))
```

```
(defun  rev (x z)
  (cond ((null x) z)
        (t (rev (cdr x) (cons (car x) z))) ))
```

```
(defun  my-append (x z)
  (cond ((null x) z)
        (t (cons (car x) (my-append (cdr x) z))) ))
```

Functions on Lists (cont.)

```
(defun  my-mapcar (f x)
  (cond ((null x) '())
        (t (cons (funcall f (car x))
                 (my-mapcar f (cdr x)))) ))
```

```
(defun  my-remove-if (f x)
  (cond ((null x) '())
        ((funcall f (car x)) (my-remove-if f (cdr x)))
        (t (cons (car x) (my-remove-if f (cdr x)))) ))
```

```
(defun  my-reduce (f x v)
  (cond ((null x) v)
        (t (funcall f (car x) (my-reduce f (cdr x) v))) ))
```

Flattening a List

We get a *flattened* form of a list if we ignore all but the initial opening and final closing parentheses in the written representation of a list.

```
> (defun flatten (x)
    (cond ((null x) x)
          ((not (consp x)) (list x))
          (t (append (flatten (car x))
                     (flatten (cdr x)) ))))
FLATTEN
```

Function `flatten` constructs a flattened list by flattening the `car` and flattening the `cdr` of a list and appending the resulting sublists.

Flattening a List (cont.)

```
> (flatten '((a) ((b b)) (((c c c)))))
(A B B C C C)
> (flatten '(1 (2 3) ((4 5 6))))
(1 2 3 4 5 6)
```

Association Lists

- An *association list*, or simply *a-list*, is a list of pairs.
- Association lists are a traditional implementation of dictionaries and environments, which map a key to an associated value.

```
> (defun bind (keys values env)
    (cons (list keys values) env))
BIND
> (bind 'a '1 '())
((A 1))
```

Association Lists (cont.)

```
> (defun bind-all (keys values env)
    (append (mapcar #'list keys values) env))
BIND-ALL
> (bind-all '(a b c) '(1 2 3) '())
((A 1) (B 2) (C 3))

> (assoc 'a '((a 1) (b 2) (c 3)))
(A 1)
> (assoc 'c '((a 1) (b 2) (c 3)))
(C 3)
```

Lists of Expressions

Lisp dialects allow $+$ and $*$ to take a list of arguments.

```
> (+ 2 3)
5
> (+ 2 3 5)
10
> (+ 2)
2
> (* 2)
2
> (+)
0
> (*)
1
```

5 An Application: Differentiation

A Differentiation Function

fun $d(x, E) =$
  if E is a constant **then** ...
  else if E is a variable **then** ...
  else if E is the sum $E_1 + E_2 + \cdots + E_k$ **then** ...
  else if E is the product $E_1 * E_2 * \cdots * E_k$ **then** ...

```
(defun d (x E)
  (cond ((constant? E) (diff-constant x E))
        ((variable? E) (diff-variable x E))
        ((sum? E) (diff-sum x E))
        ((product? E) (diff-product x E))
        (t (error "d: cannot parse ~S" E)) ))
```

Differentiation of Constants and Variables

```
(defun constant? (x) (numberp x))

(defun diff-constant (x E) 0)

(defun variable? (x) (symbolp x))

(defun diff-variable (x E)
  (if (equal x E) 1 0) )
```

Differentiation of Sums

```
(defun sum? (E)
  (and (consp E)
       (equal '+ (car E)) ))

(defun args (E) (cdr E))

(defun make-sum (x) (cons '+ x))

(defun diff-sum (x E)
  (make-sum (mapcar
              (lambda (expr) (d x expr))
              (args E)) ))
```

Differentiation of Products

```
(defun product? (E)
  (and (consp E)
       (equal '* (car E)) ))
```

```
(defun diff-product (x E)
  (let* ((arg-list (args E))
         (nargs (length arg-list)))
        (cond ((equal 0 nargs) 0)
              ((equal 1 nargs) (d x (car arg-list)))
              (t (diff-product-args x arg-list))) ))
```

Differentiation of Products (cont.)

$d(x, E_1 * E_P) = d(x, E_1) * E_P + E_1 * d(x, E_P)$ where $E_P = E_2 * \cdots * E_k$

```
(defun make-product (x) (cons '* x))
```

```
(defun diff-product-args (x arg-list)
  (let* ((E1 (car arg-list))
         (EP (make-product (cdr arg-list)))
         (DE1 (d x E1))
         (DEP (d x EP))
         (term1 (make-product (list DE1 EP)))
         (term2 (make-product (list E1 DEP))))
        (make-sum (list term1 term2)) ))
```

Using the Differentiation Function

```
> (d 'v 'v)
1

> (d 'v 'w)
0

> (d 'v '(+ u v w))
(+ 0 1 0)

> (d 'v '(* v (+ u v w)))
(+ (* 1 (* (+ U V W))) (* V (+ 0 1 0)))
```

6 Simplification of Expressions

Simplification of Expressions

- The result of the differentiation function can be made more readable by removing occurrences of 0 from sums, occurrences of 1 from products, "flattening" sums and products, etc.
- We shall implement a function `simplify` that accomplishes the simplification task.

```
> (simplify '(+ 0 1 0) )
1
> (simplify (d 'v '(+ u v w)) )
1
> (simplify '(+ (* 1 (* (+ u v w)))
               (* v (+ 0 1 0))) )
(+ U V W V)
> (simplify (d 'v '(* v (+ u v w))) )
(+ U V W V)
```

Simplification of Expressions (cont.)

```
(defun simplify (E)
  (cond ((sum? E) (simplify-sum E))
        ((product? E) (simplify-product E))
        (t E) ))
```

```
(defun simplify-sum (E)
  (simpl #'sum? #'make-sum 0 E))
```

```
(defun simplify-product (E)
  (simpl #'product? #'make-product 1 E))
```

Simplification of Expressions (cont.)

```
(defun simpl (op? make-op id E)
  (let* ((u (args E))
         (v (mapcar #'simplify u))
         (w (flat op? v))
         (x (remove-if
              (lambda (z) (equal id z))
              w))
         (y (proper make-op id x)) )
        y ))
```

Simplification of Expressions (cont.)

```
> (simplify '(* 1 (* a (+ 0 b 0))) )
(* A B)
> (simpl #'product? #'make-product 1
    '(* 1 (* a (+ 0 b 0))) )
(* A B)
> (args '(* 1 (* a (+ 0 b 0))))
(1 (* A (+ 0 B 0)))
> (mapcar #'simplify '(1 (* a (+ 0 b 0))) )
(1 (* A B))
> (flat #'product? '(1 (* a b)) )
(1 A B)
> (remove-if (lambda (z) (equal 1 z)) '(1 a b))
(A B)
> (proper #'make-product 1 '(a b))
(* A B)
```

Simplification of Expressions (cont.)

```
(defun flat (f x)
  (cond ((null x) '())
        ((not (consp x)) (list x))
        ((funcall f (car x))
         (append (flat f (args (car x)))
                 (flat f (cdr x))) )
        (t (cons (car x) (flat f (cdr x)))) ))

> (flat #'sum? '(2 (+ 3 4) 5 (* 6 7)))
(2 3 4 5 (* 6 7))
```

Simplification of Expressions (cont.)

```
(defun proper (make-op id x)
  (cond ((null x) id)
        ((null (cdr x)) (car x))
        (t (funcall make-op x)) ))

> (proper #'make-product 1 '(a b))
(* A B)
> (proper #'make-product 1 '())
1
```

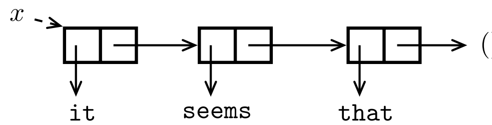
7 Storage Allocation for Lists

Storage Allocation for Lists

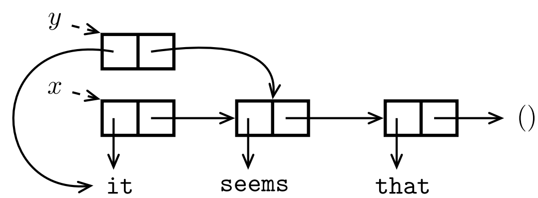
- Lists are built out of *cells* capable of holding pointers to the head and tail, or `car` and `cdr`, respectively of a list.
- The **car** operation is named after "Contents of the Address part of Register" and **cdr** is named after "Contents of the Decrement part of Register." A word in the IBM 704 could hold two pointers in the fields called the *address* part and the *decrement* part.
- When Lisp was first implemented on the IBM 704, the `cons` operation allocated a word and stuffed pointers to the head and tail in the address and decrement parts, respectively.
- The empty list `()` is a special pointer (a special address that is not used for anything else).

Storage Allocation for Lists (cont.)

```
(setq x '(it seems that))
```




```
(setq y (cons (car x) (cdr x)))
```



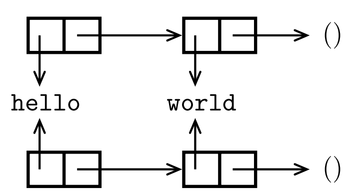
Equality

The `eq` function checks whether its two arguments are identical pointers, while the `equal` function recursively checks whether its two arguments are lists with "equal" elements.

```
> (equal 'hello 'hello)
T
> (eq 'hello 'hello)
T

> (equal '(hello world) '(hello world))
T
> (eq '(hello world) '(hello world))
NIL
```

Equality (cont.)



These two lists, though with the same elements, are allocated in different locations (and hence must be pointed to using different pointers).

Equality (cont.)

```
> (setq x '(it seems that))
(IT SEEMS THAT)
> (setq y (cons (car x) (cdr x)))
(IT SEEMS THAT)
```

```
> (equal x y)
T
> (eq x y)
NIL
```

Allocation and Deallocation

- Cells that are no longer in use have to be recovered or deallocated.
- A standard technique for allocating and deallocating cells is to link them on a list called a *free* list.
- A language implementation performs *garbage collection* when it returns cells to the free list automatically.
- When should garbage collection be performed?
  - *Lazy approach*

    Wait until memory runs out and only then collect dead cells.
  - *Eager approach*

    Each time a cell is reached, check whether the cell will be needed after the operation.

Mark-Sweep Garbage Collection

- The mark-sweep approach consists of two phases:
  - *Mark phase*

    Mark all the cells that can be reached by following the pointers.
  - *Sweep phase*

    Sweep through memory, looking for unmarked cells. Unmarked cells are returned to the free list.
- A *copying collector* avoids the expense of the sweep phase by dividing memory into two halves, the *working* half and the *free* half.
  - Cells are allocated from the working half.
  - When the working half fills up, the reachable cells are copied into consecutive locations in the free half.
  - The roles of the free and working halves are then switched.