Functional Programming in LISP

Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University

March, 2001

LISP

The first language to provide recursion, first-class function, garbage collection, and a formal language definition (in LISP itself).
LISP has been called “Lots of Silly Parentheses”.
LISP programs are untyped.
LISP implementations have been inefficient in the past.
DYNAMIC VS. STATIC SCOPE

Use dynamic scope, which leave a program sensitive to the choice of a local names within functions.

```lisp
begin
  integer x := 0;
  procedure P;
    begin print(x); end;
  procedure Q;
    begin integer x := 1; P; end;
  Q
end
```

static scope: print 0;
dynamic scope: print 1;

SCHEME: A DIALECT OF LISP

Design by Steele and Sussman, 1975. Scheme is a relatively small language that provides constructs at the core of Lisp.

Use lexical scope and supports true first-class functions.

**Interactions with Scheme interpreter :**
- Supply an expression to be evaluated.
- Bind a name to a value.

```lisp
3.14159 ; a number evaluates to itself
3.14159
(define pi 3.14159); bind a variable to a value pi

pi ; a variable evaluates to its value 3.14159
pi ; pi and pI are the same name 3.14159
```
COMMON LISP

Common LISP was created in an effort to combine the features of several dialects of LISP developed in the early 1980s, including Scheme. It is a large and complex language. Its basis, however, is pure LISP.

Recognizing the flexibility provided by dynamic scoping as well as the simplicity of static scoping, Common LISP allows both. The default is static, but by declaring a variable to be special, that variable becomes dynamically scoped.

SCHEME vs. COMMON LISP

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Common Lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define pi 3.14159)</td>
<td>(setq pi 3.14159)</td>
</tr>
<tr>
<td>(define (sq x) (* x x))</td>
<td>(defun sq (x) (* x x))</td>
</tr>
<tr>
<td>((lambda (x) (* x x)) 5)</td>
<td>((lambda (x) (* x x)) 5)</td>
</tr>
<tr>
<td>#t</td>
<td>t</td>
</tr>
<tr>
<td>#f</td>
<td>() or nil</td>
</tr>
<tr>
<td>number?</td>
<td>numberp</td>
</tr>
<tr>
<td>symbol?</td>
<td>symbolp</td>
</tr>
<tr>
<td>equal?</td>
<td>equal</td>
</tr>
<tr>
<td>null?</td>
<td>null</td>
</tr>
<tr>
<td>pair?</td>
<td>consp</td>
</tr>
<tr>
<td>(map sq '(1 2 3))</td>
<td>(mapcar (function sq) '(1 2 3))</td>
</tr>
<tr>
<td></td>
<td>or (mapcar #'sq '(1 2 3))</td>
</tr>
<tr>
<td>(map list '(a b c) '(1 2 3))</td>
<td>(mapcar #'list '(a b c) '(1 2 3))</td>
</tr>
</tbody>
</table>
SCHEME vs. COMMON LISP (cont.)

When \( f \) is a formal argument representing an \( n \)-ary function, the Scheme expression \((f \ E_1 \ E_2 \ldots \ E_n)\) translates into \((\text{funcall} \ f \ E_1 \ E_2 \ldots \ E_n)\) in Common Lisp.

There is no Common Lisp counterpart of the Scheme expression \((\text{define} \ sq \ (\lambda (x) \ (\times x \ x)))\).

LISTS

Programs and data look alike in Lisp dialects. Both are represented as lists.

A list is a sequence of zero or more values enclosed by a pair of parentheses.

\[
( )
\]
\[
\text{(it seems that)}
\]
\[
((\text{it seems that}) \ \text{you} \ (\text{like}) \ \text{me})
\]
LISTS (cont.)

(it seems that you like me)

In Lisp dialects, (+ 2 3) is an expression and a list. '(+ 2 3) is a list.

'(no quotes at (nested levels))
(no quotes at (nested levels))
⇒ (null ? ( ))
# t
⇒ (null ? nil); nil needs not be a synonym for ().
# f
⇒ (cons 'it (cons 'seems (cons 'that '( ))))
(it seems that)
⇒ (list 'it 'seems 'that)
(it seems that)
⇒ (cons it (seems))
(it seems)
⇒ (it . (seems))
(it seems)
⇒ ('(it . (seems . (that . ( )))))
(it seems that)
### OPERATIONS ON LISTS

The operations on lists are:

- `(null? X)` True if `X` is the empty list and false otherwise.
- `(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`

### LISTS IN SCHEME (cont.)

<table>
<thead>
<tr>
<th>EXPRESSION</th>
<th>SHORTHAND</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td><code>x</code></td>
<td>((it seems that) you (like) me)</td>
</tr>
<tr>
<td><code>(car x)</code></td>
<td><code>(car x)</code></td>
<td>(it seems that)</td>
</tr>
<tr>
<td><code>(car (car x))</code></td>
<td><code>(caar x)</code></td>
<td>it</td>
</tr>
<tr>
<td><code>(cdr (car x))</code></td>
<td><code>(cadr x)</code></td>
<td>(seems that)</td>
</tr>
<tr>
<td><code>(cdr x)</code></td>
<td><code>(cdr x)</code></td>
<td>(you (like) me)</td>
</tr>
<tr>
<td><code>(car (cdr x))</code></td>
<td><code>(cadr x)</code></td>
<td>You</td>
</tr>
<tr>
<td><code>(cdr (cdr x))</code></td>
<td><code>(cdadr x)</code></td>
<td>((like) me)</td>
</tr>
</tbody>
</table>
### SOME SCHEME CONSTRUCTS

```
(define pi 3.14159) ; give name pi to 3.14159
(define (sq x) (* x x)) ; fun sq(x) = x * x
(define sq (lambda (x) (* x x))) ; fun sq(x) = x * x
(lambda (x) (* x x)) ; anonymous function value
(* E₁ E₂) ; E₁ * E₂
(E₁ E₂ E₃) ; apply he value of E₁ as a function to arguments E₂ and E₃
(if P E₁ E₂) ; if P then E₁ then E₂
(cond (P₁ E₂₁) (P₂ E₂₂) (else E₃)) ; else if P₂ then E₂₂ ; else E₃
```

### SOME SCHEME CONSTRUCTS (cont.)

```
(let ((X₁ E₁) (X₂ E₂)) E₃) ; evaluate E₁ and E₂; then; evaluate E₃ with x₁ and x₂; bound to their values
(let* ((X₁ E₁) (X₂ E₂)) E₃) ; let val x₁ = E₁; val x₂ = E₂; in E₃; end
(quote blue) ; symbol blue
(quote (blue green red)) ; list (blue green red)
(list E₁ E₂ E₃) ; list of the values of E₁, E₂, E₃
```
FUNCTIONS ON LISTS

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

(define rev (x z)
  (cond ((null? x) z)
        (else (rev (cdr x) (cons (car x) z)))))

(define append (x z)
  (cond ((null? x) z)
        (else (cons (car x) (append (cdr x) z)))))

(define map (f x)
  (cond ((null? x) '())
        (else (cons (f (car x)) (map f (cdr x))))))

---

FUNCTIONS ON LISTS (cont.)

(define remove-if (f x)
  (cond ((null? x) '()) ((f (car x))
                       (remove-if f (cdr x)))
        (else (cons (car x) (remove-if f (cdr x))))))

(define reduce (f x v)
  (cond ((null? x) v)
        (else (f (car x) (reduce f (cdr x) v)))))
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.

```
> (define bind (keys values env)
   (cons (list keys values) env) )
BIND
> (bind 'a '1 '())
  ((a 1))
> (define bind-all (keys values env)
   (append (map list keys values) env) )
```

ASSOCIATION LISTS (cont.)

```
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)
```
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.

> (flat '((a) ((b b)) (((c c c)))))
(a b b c c c)

> (flat '(1 (2 3) ((4 5 6)))))
(1 2 3 4 5 6)

(define flat x)
  (cond ((null? x) '())
    ((not (pair? x)) (list x))
    (else (append (flat (car x))
                 (flat (cdr x)) ))))

"Lots of Silly Parentheses"?

LISTS OF SUBEXPRESSIONS

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

> (+ 2 3)
5
> (+ 2 3 5)
10
> (+ 2)
2
> (* 2)
2
> (+)
0
> (*)
1
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+...+E_n then ...
else if E is the product E_1*E_2*...*E_k then ...

(define 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))
        (else (error "d: cannot parse" E)) ))

DIFFERENTIATION OF CONSTANTS AND VARIABLES

(define constant? number?)
(define (diff-constant x E) 0)
(define variable? (x) (symbol? x))
(define diff-variable (x E)
  (if (equal? x E) 1 0) )
DIFFERENTIATION OF SUMS

(define sum? (E)
  (and (pair? E)
       (equal? '+ (car E)) ))

(define (args E) (cdr E))

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

(define (diff-sum x E)
  (make-sum (map
             (lambda (expr) (d x expr))
             (args E))))

DIFFERENTIATION OF PRODUCTS

(define product? E)
  (and (pair? E)
       (equal? '* (car E)) ))

(define (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)))
          (else (diff-product-args x arg-list))))

DIFFERENTIATION OF PRODUCTS (cont.)

\[ d(x, E_1 \cdot E_p) = d(x, E_1) \cdot E_p + E_1 \cdot d(x, E_p) \]

where \( E_p = E_2 \cdot \ldots \cdot E_k \)

(define (make-product x) (cons '* x))

(define (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)))
> (d 'v '(u v w))
*** - d: cannot parse (u v w)
1. Break> ^D
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.

```lisp
> (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.)

```lisp
(define (simplify E)
  (cond ((sum? E) (simplify-sum E))
        ((product? E) (simplify-product E))
        (else E )))
(define (simplify-sum E)
  (simpl sum? make-sum 0 E))
(define (simplify-product E)
  (simpl product? make-product 1 E))
(define (simpl op? make-op id E)
  (let* ((u (args E))
          (v (map 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.)

(define flat (f x)
  (cond ((null? x) '())
    ((not (pair? x)) (list x))
    ((f (car x))(append (flat f (args (car x)))
                         (flat f (cdr x))) )
    (else (cons (car x) (flat f (cdr x))))) ))

(define (proper make-op id x)
  (cond ((null? x) id)
    ((null? (cdr x)) (car x))
    (else (make-op x)) ))

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

> (proper make-product 1 '(a b))
(* a b)

> (proper make-product 1 '())
1

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.)

\[(\text{cons 'it (cons 'seems (cons 'that nil))})\]

- car returns the pointer in the first field of the cell;
- cdr return the second pointer; and cons allocates a cell.
EQUALITY

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

\begin{itemize}
\item \texttt{> (equal? 'hello 'hello)}
\item \texttt{t}
\item \texttt{> (eq? 'hello 'hello)}
\item \texttt{t}
\item \texttt{> (equal? '(hello world) '(hello world))}
\item \texttt{t}
\item \texttt{> (eq? '(hello world) '(hello world))}
\item \texttt{#f}
\end{itemize}

\begin{itemize}
\item \texttt{hello}
\item \texttt{world}
\item \texttt{()}\end{itemize}

EQUALITY (cont.)

\begin{itemize}
\item \texttt{> (define x '(it seems that))}
\item \texttt{x}
\item \texttt{> (define y (cons (car x) (cdr x)))}
\item \texttt{y}
\item \texttt{> (equal x y)}
\item \texttt{#t}
\item \texttt{> (eq x y)}
\item \texttt{#f}
\end{itemize}
**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.