Functional Programming with Lists

Scheme, a dialect of LISP

Interact with a scheme interpreter

- Online interpreter
  - https://inst.eecs.berkeley.edu/~cs61a/fa14/assets/interpreter/scheme.html
- Supply an expression to be evaluated
- Bind a name to a value
  - e.g., (define pi 3.14159)
- Names
  - Contain special characters but not parentheses
    - e.g., long-name, research!emlin, back-at-5:00pm.
  - Begin with any character but not a number
  - Ignore the distinction between uppercase and lowercase letters
    e.g., pi, Pi, pI and PI are all the same name
- Comments begin with a semicolon and continue to the end of the line

Write an expression

- Use a form of prefix notation in which parentheses surround an operator and its operands
  - Infix: 1 + 2
  - Prefix + 1 2
  - Postfix 1 2 +
- The general form of an expression in Scheme

  $(E_1\ E_2\ \ldots\ E_k)$
  - E_1: an operator
  - E_2, E_3 … E_k: operands
  - e.g.,
    - `( * 5 7 )                ; (5*7)`
    - `( + 4 ( * 5 7 ) )        ; 4+(5*7)`

Define a function

- (define (<function-name> <formal-parameters>) <expression>)
  - e.g., (define (square x) (* x x))
    (square 5) ;apply function square to 5
- (define <function-name> (lambda (<formal-parameters>) <expression>))
  - e.g., (define square (lambda (x) (* x x)))
- define supports recursive functions

Define a function

- (define (<function-name> <formal-parameters>) <expression>)
  - e.g., (define (mult x y) (* x y))
    (mult 5 7) ;apply function square to 5
- (define <function-name> (lambda (<formal-parameters>) <expression>))
  - e.g., (define mult (lambda (x y) (* x y)))
- define supports recursive functions

Define a function

- Anonymous function values
    - (lambda (<formal-parameters>) <expression>)
        - e.g., ((lambda (x) (* x x)) 5)
          ;unnamed function applied to 5
    - Can appear within expressions, either as an operator or as an argument
    - Recursion is not supported directly

Conditions

- Predicates
  - number? / symbol? / equal?
- If
  - (if P E_1 E_2) ; if P then E_1 else E_2
- Cond
  - (cond (P_1 E_1) ; if P_1 then E_1
    … ; …
    (P_k E_k) ; else if P_k then E_k
    (else E_{k+1})) ; else E_{k+1}

Quoting

- A quoted item evaluates to itself
- Quoting is used to choose whether spelling is treated as a symbol of a variable name
  - e.g.,
    - pi ; variable name, bound to 3.141592
    - ‘pi ; the spelling of the symbol
  - e.g.,
    - (define f *) ; * represent the multiplication function
    - (define f ‘*) ; ‘* represent the symbol *

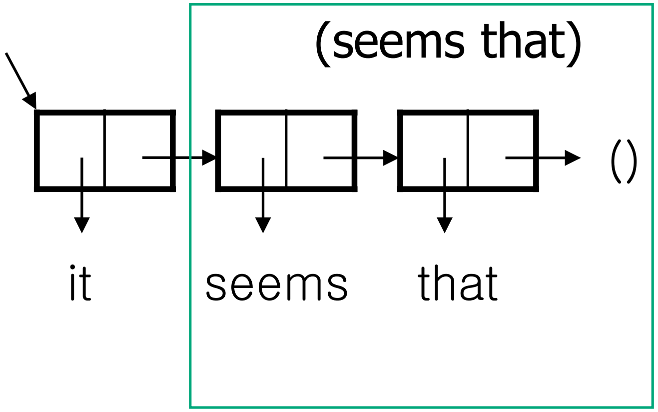
The Structure of Lists

List Element

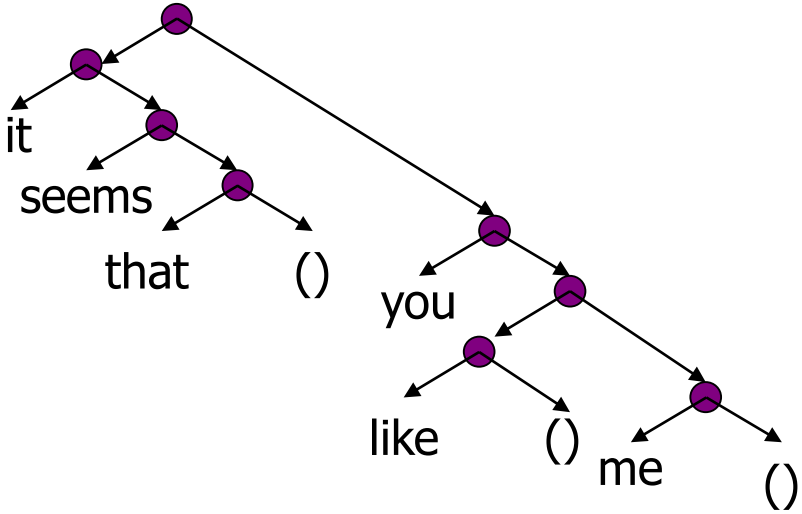
- List
  - A sequence of zero or more values
  - Potential list elements : booleans, numbers, symbols, other list and functions
  - Parentheses enclose list elements
- Example
  - (): empty list (null list) with zero elements
  - (it seems that): The list has three symbols of it, seems and that.

Examples of lists

- Structure of lists



(it seems that)



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

Expression and List

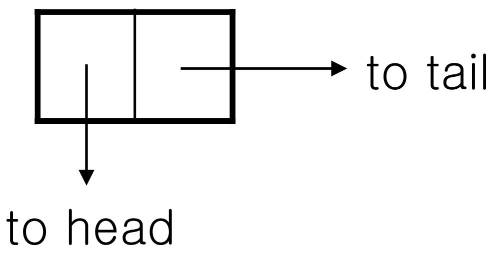
- Is (+ 2 3) an expression or a list ?
  - The Answer is both.
  - The Scheme interpreter treats it as an expression.
- Quoting tells the interpreter to treat (+ 2 3) as a list
  - (+ 2 3) -> expression
    - Result : 5
  - ‘(+ 2 3) -> list
    - Result : (+ 2 3)
  - Single quote is sufficient to say that the construct immediately following the quote stands for itself.
  - e.g., ‘(no quotes at (nested levels))
    - Result : (no quotes at (nested levels))

Operations on Lists

- Basic operations on lists
  - (null? x) : True if x is the empty list and false otherwise
    - (null? ()) -> #t -> empty list
    - (null? nil) -> #f -> not empty list
    - nil need not be synonym for ()
  - (car x) : The first element of a nonempty list x
    - (car ‘(a b c)) -> a
    - a is an element, not list. (a) is a list which has one element a
  - (cdr x) : The rest of the list x after the first element is removed
    - (cdr ‘(a b c)) -> (b c)
  - (cons a x) : A value with car a and cdr x; that is,
    - (cons ‘a ‘(b c)) -> (a b c)
    - (car (cons ‘a ‘(b c))) -> a
    - (cdr (cons ‘a ‘(b c))) -> (b c)
    - (cons ‘d ‘(a b c)) -> (d a b c)

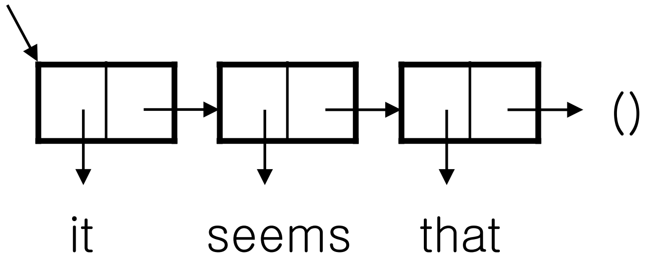
Storage Allocation For Lists

- A list is made up of cells.
- A cells with pointers to the head and tail of a list.

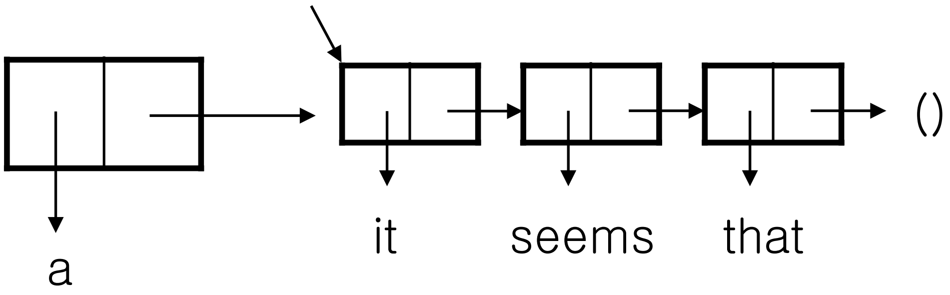


- *Cons* allocates a single cell

- (cons ‘a ‘(it seems that))
  - (a it seems that)



(it seems that)



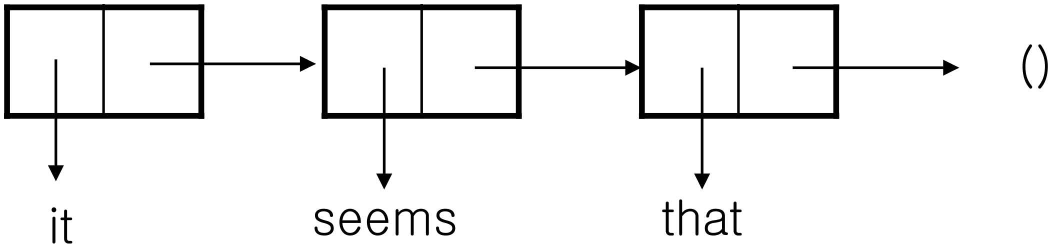
(a it seems that)

Cons Allocates Cells (1)

- Lists are built out of cells capable of holding pointers to the head(car) and tail(cdr) of a list.
    - car : “Contents of the Address part of Register”
    - cdr : “Contents of the Decrement part of Register”
- Cons : allocates a word and stuffed pointers to the car and cdr of a list.

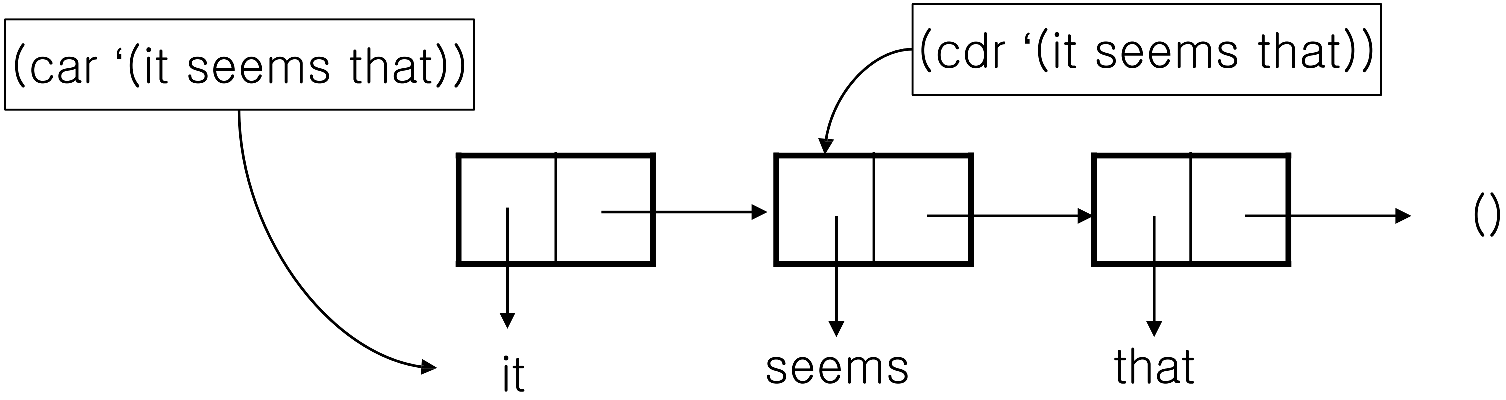
Cons Allocates Cells (2)

- The empty list () is a special pointer.
  - Think of () as a special address that is not used for anything else.
- (cons ‘it (cons ‘seems (cons ‘that ‘())))



Cons Allocates Cells (3)

- null? : compares its argument for equality with ().
- car : returns the pointer in the first field.
- cdr : returns the pointer in the second field.



How to Build lists

- Cons operation builds
  - (cons a x) create a value with head a and tail x
  - Alternative “dotted” notation for (cons a x) is (a . x)
  - More precisely, a cons operation builds a pair from its operands.
- The name ‘list’ is reserved for a chain of pairs ending in an empty list.
  - Repeated application of cdr eventually results in the empty list ().
- (cons ‘it (cons ‘seems (cons ‘that ‘())))
- = ‘(it . (seems . (that . ()))) = (list ‘it ‘seems ‘that)
  - Result : (it seems that)
  - (that . ()) = (that)
  - (seems . (that . ())) = (seems that)

Practice for Scheme

To define cadr function

- (define (square x) (* x x))

  (square 5) ;apply function square to 5
- Predicates
  - null? ; empty list?
- Cond
  - (cond (P_1 E_1) ; if P_1 then E_1

    … ; …

    (P_k E_k) ; else if P_k then E_k

    (else E_{k+1})) ; else E_{k+1}

```
(define (cadr List)
  (cond ((null? List) (display 'error))
      ((null? (cdr List)) (display 'error))
      (else (car (cdr List)))))
```

- (cadr ‘(2 4 6 1)) : 4

cadr

```
(define (cadr List)
  (cond ((null? List) (display 'error))
      ((null? (cdr List)) (display 'error))
      (else (car (cdr List)))))
```

- (cadr ‘(2 4 6 1)) : 4

Find the second element in a list

```
(define (second List)
  (cond ((null? List) (display 'error))
      ((null? (cdr List)) (display 'error))
      (else (cadr List))))
```

- (second ‘(2 4 6 1)) : 4

Find the last element in a list

```
(define (last List)
  (cond ((null? List) (display 'error))
      ((null? (cdr List)) (car List))
      (else (last (cdr List)))))
```

- (last '(2 4 6 1)) : 1

Find the length of a list

```
(define (length List)
  (if (null? List) 0
    (+ 1 (length (cdr List)))))
```

- (length ‘(2 4 6 1)) : 4

Find the sum of elements in a list

```
(define (sum List)
  (if (null? List) 0
      (+ (car List) (sum (cdr List)))))
```

- (sum ‘(2 4 6 1)) : 13

Find the maximum value in a list

```
(define (maximum List)
  (cond ((null? List) (display ‘error))
        ((null? (cdr List)) (car List))
        (else (max (car List)
          (maximum (cdr List))))))
(define (max x y)
  (if (> x y) x y))
```

- (maximum ‘(2 4 6 1)) : 6

Find the minimum value in a list

```
(define (minimum List)
  (cond ((null? List) (display ‘error))
        ((null? (cdr List)) (car List))
        (else (min (car List)
          (minimum (cdr List))))))
(define (min x y)
  (if (< x y) x y))
```

- (minimum ‘(2 4 6 1)) : 1