

CSc 372

Comparative Programming Languages

38 : Scheme — List Processing

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2013 Christian Collberg

Constructing Lists

- The most important data structure in Scheme is the list.
- Lists are constructed using the function `cons`:

```
(cons first rest)
```

`cons` returns a list where the first element is `first`, followed by the elements from the list `rest`.

```
> (cons 'a '())
```

```
(a)
```

```
> (cons 'a (cons 'b '()))
```

```
(a b)
```

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

```
(a b c)
```

Constructing Lists...

- There are a variety of short-hands for constructing lists.
- Lists are **heterogeneous**, they can contain elements of different types, including other lists.

```
> '(a b c)
```

```
(a b c)
```

```
> (list 'a 'b 'c)
```

```
(a b c)
```

```
> '(1 a "hello")
```

```
(1 a "hello")
```

Examining Lists

- `(car L)` returns the first element of a list. Some implementations also define this as `(first L)`.
- `(cdr L)` returns the list L, without the first element. Some implementations also define this as `(rest L)`.
- Note that `car` and `cdr` do not destroy the list, just return its parts.

```
> (car '(a b c))  
'a  
> (cdr '(a b c))  
'(b c)
```

Examining Lists...

- Note that (cdr L) always returns a list.

```
> (car (cdr '(a b c)))
```

```
'b
```

```
> (cdr '(a b c))
```

```
'(b c)
```

```
> (cdr (cdr '(a b c)))
```

```
'(c)
```

```
> (cdr (cdr (cdr '(a b c))))
```

```
'()
```

```
> (cdr (cdr (cdr (cdr '(a b c)))))
```

```
error
```

Examining Lists...

- A shorthand has been developed for looking deep into a list:

```
(clist of "a" and "d"r L)
```

Each "a" stands for a car, each "d" for a cdr.

- For example, (caddar L) stands for

```
(car (cdr (cdr (car L))))
```

```
> (cadr '(a b c))
```

```
'b
```

```
> (cddr '(a b c))
```

```
'(c)
```

```
> (caddr '(a b c))
```

```
'c
```

Lists of Lists

- Any S-expression is a valid list in Scheme.
- That is, lists can contain lists, which can contain lists, which...

```
> '(a (b c))
```

```
(a (b c))
```

```
> '(1 "hello" ("bye" 1/4 (apple)))
```

```
(1 "hello" ("bye" 1/4 (apple)))
```

```
> (caaddr '(1 "hello" ("bye" 1/4 (apple))))
```

```
"bye"
```

List Equivalence

- `(equal? L1 L2)` does a structural comparison of two lists, returning `#t` if they “look the same”.
- `(equiv? L1 L2)` does a “pointer comparison”, returning `#t` if two lists are “the same object”.

```
> (equiv? '(a b c) '(a b c))  
false  
> (equal? '(a b c) '(a b c))  
true
```


List Equivalence. . .

- This is sometimes referred to as **deep equivalence** vs. **shallow equivalence**.

```
> (define myList '(a b c))
```

```
> (equiv? myList myList)
```

```
true
```

```
> (equiv? '(a (b c (d))) '(a (b c (d))))
```

```
false
```

```
> (equal? '(a (b c (d))) '(a (b c (d))))
```

```
true
```

Predicates on Lists

- `(null? L)` returns `#t` for an empty list.
- `(list? L)` returns `#t` if the argument is a list.

```
> (null? '())  
#t  
> (null? '(a b c))  
#f  
> (list? '(a b c))  
#t  
> (list? "(a b c)")  
#f
```

List Functions — Examples...

```
> (memq 'z '(x y z w))
#t
> (car (cdr (car '((a) b (c d)))))
(c d)
> (caddr '(a) b (c d))
(c d)
> (cons 'a '())
(a)
> (cons 'd '(e))
(d e)
> (cons '(a b) '(c d))
((a b) (c d))
```

Recursion over Lists — cdr-recursion

- Compute the length of a list.
- This is called **cdr-recursion**.

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

```
> (length '(1 2 3))
```

```
3
```

```
> (length '(a (b c) (d e f)))
```

```
3
```

Recursion over Lists — car-cdr-recursion

- Count the number of atoms in an S-expression.
- This is called **car-cdr-recursion**.

```
(define (atomcount x)
  (cond
    [(null? x) 0]
    [(list? x)
     (+ (atomcount (car x))
        (atomcount (cdr x)))]
    [else 1]
  ))
```

```
> (atomcount '(1))
```

```
1
```

```
> (atomcount '("hello" a b (c 1 (d))))
```

```
6
```

Recursion Over Lists — Returning a List

- Map a list of numbers to a new list of their absolute values.
- In the previous examples we returned an atom — here we're mapping a list to a new list.

```
(define (abs-list L)
  (cond
    [(null? L) '()]
    [else (cons (abs (car L))
                 (abs-list (cdr L)))]
  )
)
```

```
> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
```

Recursion Over Two Lists

- `(atom-list-eq? L1 L2)` returns `#t` if `L1` and `L2` are the same list of atoms.

```
(define (atom-list-eq? L1 L2)
  (cond
    [(and (null? L1) (null? L2)) #t]
    [(or (null? L1) (null? L2)) #f]
    [else (and
            (atom? (car L1))
            (atom? (car L2))
            (eqv? (car L1) (car L2))
            (atom-list-eq? (cdr L1) (cdr L2)))]
  )
)
```

Recursion Over Two Lists...

```
> (atom-list-eq? '(1 2 3) '(1 2 3))
```

```
#t
```

```
> (atom-list-eq? '(1 2 3) '(1 2 a))
```

```
#f
```


Append

```
(define (append L1 L2)
  (cond
    [(null? L1) L2]
    [else
     (cons (car L1)
           (append (cdr L1) L2))])
)
```

```
> (append '(1 2) '(3 4))
```

```
(1 2 3 4)
```

```
> (append '() '(3 4))
```

```
(3 4)
```

```
> (append '(1 2) '())
```

```
(1 2)
```

Deep Recursion — equal?

```
(define (equal? x y)
  (or (and (atom? x) (atom? y) (eq? x y))
      (and (not (atom? x))
            (not (atom? y))
            (equal? (car x) (car y))
            (equal? (cdr x) (cdr y)))))
```

```
> (equal? 'a 'a)
```

```
#t
```

```
> (equal? '(a) '(a))
```

```
#t
```

```
> (equal? '((a)) '((a)))
```

```
#t
```

Patterns of Recursion — cdr-recursion

- We process the elements of the list one at a time.
- Nested lists are not descended into.

```
(define (fun L)
  (cond
    [(null? L) return-value]
    [else ... (car L) ... (fun (cdr L)) ...]
  )
)
```

Patterns of Recursion — car-cdr-recursion

- We descend into nested lists, processing every atom.

```
(define (fun x)
  (cond
    [(null? x) return-value]
    [(atom? x) return-value]
    [(list? x)
     ... (fun (car x)) ...
     ... (fun (cdr x)) ...]
    [else return-value]
  ))
```

Patterns of Recursion — Maps

- Here we map one list to another.

```
(define (map L)
  (cond
    [(null? L) '()]
    [else (cons (...(car L) ...)
                 (map (cdr L)))]
  )
)
```

Example: Binary Trees

- A binary tree can be represented as nested lists:

(4 (2 () ()) (6 (5 () ()) ()))

- Each node is represented by a triple

(data left-subtree right-subtree)

- Empty subtrees are represented by ().

Example: Binary Trees...

```
(define (key tree) (car tree))
(define (left tree) (cadr tree))
(define (right tree) (caddr tree))

(define (print-spaces N)
  (cond
    [(= N 0) ""]
    [else (begin
             (display " ")
             (print-spaces (- N 1)))]))

(define (print-tree tree)
  (print-tree-rec tree 0))
```

Example: Binary Trees...

```
(define (print-tree-rec tree D)
  (cond
    [(null? tree)]
    [else (begin
             (print-spaces D)
             (display (key tree)) (newline)
             (print-tree-rec (left tree) (+ D 1))
             (print-tree-rec (right tree) (+ D 1))
            )]])
```

```
> (print-tree '(4 (2 () ()) (6 (5 () ()) ())))
```

```
4
```

```
  2
```

```
  6
```

```
    5
```


Binary Trees using Structures

- We can use structures to define tree nodes.

```
(define-struct node (data left right))

(define (tree-member x T)
  (cond
    [(null? T) #f]
    [(= x (node-data T)) #t]
    [< x (node-data T)]
      (tree-member x (node-left T))]
    [else
      (tree-member x (node-right T))]
  )
)
```

Binary Trees using Structures...

```
(define tree
  (make-node 4
    (make-node 2 '() '())
    (make-node 6
      (make-node 5 '() '())
      (make-node 9 '() '()))))
```

```
> (tree-member 4 tree)
```

```
true
```

```
> (tree-member 5 tree)
```

```
true
```

```
> (tree-member 19 tree)
```

```
false
```

Homework

- Write a function `swapFirstTwo` which swaps the first two elements of a list. Example: $(1\ 2\ 3\ 4) \Rightarrow (2\ 1\ 3\ 4)$.
- Write a function `swapTwoInLists` which, given a list of lists, forms a new list of all elements in all lists, with first two of each swapped. Example: $((1\ 2\ 3)\ (4)\ (5\ 6)) \Rightarrow (2\ 1\ 3\ 4\ 6\ 5)$.