1 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)
```

2 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")
```
3 Examining Lists

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

\[
\begin{align*}
> \text{car } '(a \ b \ c) \\
' a \\
> \text{cdr } '(a \ b \ c) \\
'(b \ c)
\end{align*}
\]

4 Examining Lists...

- Note that \( \text{cdr} \ L \) always returns a list.

\[
\begin{align*}
> \text{car } ((\text{cdr } '(a \ b \ c))) \\
' b \\
> \text{cdr } '((a \ b \ c)) \\
'(b \ c) \\
> \text{cdr } (\text{cdr } (\text{cdr } '((a \ b \ c)))) \\
'(c) \\
> \text{cdr } (\text{cdr } (\text{cdr } (\text{cdr } '((a \ b \ c))))) \\
'() \\
> \text{cdr } (\text{cdr } (\text{cdr } (\text{cdr } '((a \ b \ c))))) \\
\text{error}
\end{align*}
\]

5 Examining Lists...

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

\[
\text{(clist of "a" and "d"r L)}
\]

Each "a" stands for a \text{car}, each "d" for a \text{cdr}.
- For example, \( \text{caddr} \ L \) stands for

\[
\text{(car } (\text{cdr } (\text{cdr } (\text{car } L))))
\]

\[
\begin{align*}
> \text{cadr } '((a \ b \ c)) \\
'b \\
> \text{cdadr } '((a \ b \ c)) \\
'(c) \\
> \text{caddadr } '((a \ b \ c)) \\
'c
\end{align*}
\]
6 Lists of Lists

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

\[
\begin{align*}
> & ' (a (b c)) \\
& (a (b c)) \\
> & '(1 "hello" ("bye" 1/4 (apple))) \\
& (1 "hello" ("bye" 1/4 (apple))) \\
> & (caraddr '(1 "hello" ("bye" 1/4 (apple)))) \\
& "bye"
\end{align*}
\]

7 List Equivalence

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

\[
\begin{align*}
> & (eqv? '(a b c) '(a b c)) \\
& \text{false} \\
> & (equal? '(a b c) '(a b c)) \\
& \text{true}
\end{align*}
\]

8 List Equivalence...

- This is sometimes referred to as \textit{deep equivalence} vs. \textit{shallow equivalence}.

\[
\begin{align*}
> & (define myList '(a b c)) \\
> & (eqv? myList myList) \\
& \text{true} \\
> & (eqv? '(a (b c (d))) '(a (b c (d)))) \\
& \text{false} \\
> & (equal? '(a (b c (d))) '(a (b c (d)))) \\
& \text{true}
\end{align*}
\]

9 Predicates on Lists

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

\[
\begin{align*}
> & (null? '()) \\
& \text{#t} \\
> & (null? '(a b c)) \\
& \text{#f} \\
> & (list? '(a b c)) \\
& \text{#t} \\
> & (list? "(a b c)"
& \text{#f}
\end{align*}
\]
10 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))

11 Recursion over Lists — cdr-recursion

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

\[
\begin{align*}
\text{(define (length x)} & \\
\text{ (cond} & \\
\text{ \quad [(null? x) 0]} & \\
\text{ \quad [else (+ 1 (length (cdr x)))]} & \\
\text{ \quad ])} & \\
\text{)} & \\
> (length '(1 2 3)) & \\
3 & \\
> (length '(a (b c) (d e f))) & \\
3 & \\
\end{align*}
\]

12 Recursion over Lists — car-cdr-recursion

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

\[
\begin{align*}
\text{(define (atomcount x)} & \\
\text{ (cond} & \\
\text{ \quad [(null? x) 0]} & \\
\text{ \quad [(list? x]} & \\
\text{ \quad \quad (+ (atomcount (car x))]} & \\
\text{ \quad \quad (atomcount (cdr x)))]} & \\
\text{ \quad [else 1]} & \\
\text{ \quad ])} & \\
\text{)} & \\
> (atomcount '(1)) & \\
1 & \\
> (atomcount '("hello" a b (c 1 (d)))) & \\
6 & \\
\end{align*}
\]
13 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)

14 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)))]
  )
)
```

15 Recursion Over Two Lists...

> (atom-list-eq? '(1 2 3) '(1 2 3))
#t
> (atom-list-eq? '(1 2 3) '(1 2 a))
#f
16 Append

\[
\text{(define (append L1 L2)} \\
\quad \text{(cond)} \\
\quad \quad [\text{(null? L1)} \text{ L2}] \\
\quad \quad [\text{else} \\
\quad \quad \quad \text{(cons (car L1)} \\
\quad \quad \quad \quad \text{(append (cdr L1) L2)}))] \\
\quad \text{)} \\
\text{)}
\]

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

17 Deep Recursion — equal?

\[
\text{(define (equal? x y)} \\
\quad \text{(or (and (atom? x) (atom? y) (eq? x y))} \\
\quad \quad (and (not (atom? x)) \\
\quad \quad \quad (not (atom? y)) \\
\quad \quad \quad \text{(equal? (car x) (car y))} \\
\quad \quad \quad \text{(equal? (cdr x) (cdr y)))))}
\]

> (equal? 'a 'a)
#t
> (equal? '(a) '(a))
#t
> (equal? '((a)) ')((a)))
#t

18 Patterns of Recursion — cdr-recursion

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

\[
\text{(define (fun L)} \\
\quad \text{(cond)} \\
\quad \quad [\text{(null? L)} \text{ return-value}] \\
\quad \quad [\text{else} \text{ ...}(\text{fun (cdr L)}) ...] \\
\quad \text{)} \\
\text{)}
\]

19 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]
  ))

20 Patterns of Recursion — Maps

• Here we map one list to another.

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

21 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 ()

22 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))
23 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

24 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))]
  )
)

25 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
26  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: \(((1\ 2\ 3)\ (4)\ (5\ 6)) \Rightarrow (2\ 1\ 3\ 4\ 6\ 5))\).