Constructing Lists

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

\[
(\text{cons} \text{ first} \text{ rest})
\]

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

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

Examining Lists

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

\[
\begin{align*}
> & '(a\ b\ c) \\
& (a\ b\ c) \\
> & (\text{list} \ 'a\ 'b\ 'c) \\
& (a\ b\ c) \\
> & '((1\ a\ "hello") ) \\
& (1\ a\ "hello") \\
\end{align*}
\]

\[
\begin{align*}
> & (\text{car} \ (\text{list} \ 'a\ 'b\ 'c) ) \\
& 'a \\
> & (\text{cdr} \ (\text{list} \ 'a\ 'b\ 'c) ) \\
& '(b\ c)
\end{align*}
\]
Examining Lists...

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

  ```lisp
  > (\text{car} (\text{cdr} '(a b c)))
  'b
  > (\text{cdr} '(a b c))
  '(b c)
  > (\text{cdr} (\text{cdr} '(a b c)))
  '(c)
  > (\text{cdr} (\text{cdr} (\text{cdr} '(a b c))))
  ()
  > (\text{cdr} (\text{cdr} (\text{cdr} (\text{cdr} '(a b c)))))
  error
  ```

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

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

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

- For example, \((\text{caddar } L)\) stands for
  \( \text{(car (cdr (cdr (car L)))}) \)

  ```lisp
  > (\text{cadr} '(a b c))
  'b
  > (\text{cddr} '(a b c))
  '(c)
  > (\text{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...

  ```lisp
  > '(a (b c))
  (a (b c))
  > '(1 "hello" ("bye" 1/4 (apple)))
  (1 "hello" ("bye" 1/4 (apple)))
  > (\text{caaddr} '(1 "hello" ("bye" 1/4 (apple))))
  "bye"
  ```

List Equivalence

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

  ```lisp
  > (\text{eqv? } '(a b c) '(a b c))
  false
  > (\text{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))
> (eqv? myList myList)
true
> (eqv? '(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**.

```scheme
(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.

```scheme
(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)`
- `(abs-list '(1 2 3))` → `'(1 2 3)`
- `(abs-list '(1 2 a))` → `'(1 2 a)`

---

Recursion Over Two Lists

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

```scheme
(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)))]
  ))
)
```

- `(atom-list-eq? '(1 2 3) '(1 2 3))` → `#t`
- `(atom-list-eq? '(1 2 3) '(1 2 a))` → `#f`

---

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]
    [(list? L) 
      ... (fun (car L)) ... (fun (cdr L)) ...]
    [else return-value])
)

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.

```scheme
(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...

```scheme
(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-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
```

Example: Binary Trees...

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

520 — Spring 2008 — 38
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))])
)
```

```
(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) ⇒ (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)) ⇒ (2 1 3 4 6 5).`