1 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`.

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

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.

\[\begin{align*}
> '(\text{a b c}) \\
& \text{(a b c)} \\
> (\text{list } \text{a } \text{b } \text{c}) \\
& \text{(a b c)} \\
> '(1 \text{ a } \text{"hello"}) \\
& \text{(1 a "hello")} \\
\end{align*}\]
3 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.

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

4 Examining Lists...

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

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

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 car, each "d" for a cdr.

- For example, (caddar L) stands for

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

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

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

> ' (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"

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”.

> (eqv? '(a b c) '(a b c))
false

> (equal? '(a b c) '(a b c))
true

8 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

9 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
10 List Functions — Examples...

\[
\begin{align*}
> & \text{(memq 'z '(x y z w))} \\
& \text{#t} \\
> & \text{(car (cdr (car '((a) b (c d)))))} \\
& \text{(c d)} \\
> & \text{(caddr '((a) b (c d)))} \\
& \text{(c d)} \\
> & \text{(cons 'a '())} \\
& \text{(a)} \\
> & \text{(cons 'd '(e))} \\
& \text{(d e)} \\
> & \text{(cons '(a b) '(c d))} \\
& \text{((a b) (c d))}
\end{align*}
\]

11 Recursion over Lists — cdr-recursion

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

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

12 Recursion over Lists — car-cdr-recursion

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

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

\[
\begin{align*}
&(\text{define } (\text{abs-list } L) \\
&\quad (\text{cond} \\
&\quad\quad [(\text{null? } L) \ '()] \\
&\quad\quad [\text{else } (\text{cons } (\text{abs } (\text{car } L)) \ \\
&\quad\quad\quad (\text{abs-list } (\text{cdr } L)))])
\end{align*}
\]

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

14 Recursion Over Two Lists

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

\[
\begin{align*}
&(\text{define } (\text{atom-list-eq? } L1 \ L2) \\
&\quad (\text{cond} \\
&\quad\quad [(\text{and } (\text{null? } L1) \ (\text{null? } L2)) \ #t] \\
&\quad\quad [(\text{or } (\text{null? } L1) \ (\text{null? } L2)) \ #f] \\
&\quad\quad [\text{else } (\text{and} \\
&\quad\quad\quad (\text{atom? } (\text{car } L1)) \ \\
&\quad\quad\quad (\text{atom? } (\text{car } L2)) \\
&\quad\quad\quad (\text{eqv? } (\text{car } L1) (\text{car } L2)) \\
&\quad\quad\quad (\text{atom-list-eq? } (\text{cdr } L1) \ (\text{cdr } L2)))]
\end{align*}
\]

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

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

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

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

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

17 Deep Recursion — equal?

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

> (equal? 'a 'a)
#t

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

> (equal? '((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.

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

19 Patterns of Recursion — car-cdr-recursion

- We descend into nested lists, processing every atom.
\begin{verbatim}
(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))
\end{verbatim}
23 Example: Binary Trees...

```scheme
(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.

```scheme
(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...

```scheme
(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: \(((1\ 2\ 3)\ (4)\ (5\ 6)) \Rightarrow (2\ 1\ 3\ 4\ 6\ 5)\).