Lists and Recursion

CS 270 Math Foundations of CS

Jeremy Johnson

Objective

- To provide a recursive definition of lists and several recursive functions for processing lists that mimic the recursive definition

- To practice using recursive data structures and writing recursive functions in Racket

Fundamental List Functions

$L = (x_1 \dots x_n)$

null? : All → Boolean

cons? : All → Boolean

cons : All × List → Cons

(cons x L) = $(x\ x_1 \dots x_n)$

first : Cons → All

(first (cons x L)) = x

rest : Cons → List

(rest (cons x L)) = L

Building Lists

> (list x_1 … x_n) [macro – variable # of args]

(cons x_1 (cons x_2 … (cons x_n nil)…))

> (cons? ‘()) => #f

> (null? ‘()) => #t

> (cons 1 ‘()) => ‘(1)

(cons? (cons 1 ‘())) => #t

(cons 1 (cons 2 (cons 3 ‘()))) => ‘(1 2 3)

(list 1 2 3) => ‘(1 2 3)

(cons (cons 1 ‘()) ‘()) => ‘((1))

Exercise

How do you construct the list ((1) 2)?

Exercise

How do you construct the list ((1) 2)?

(cons 1 ‘()) = (1)

(cons 2 ‘()) = (2)

Want (cons x y) where x = ‘(1) and y = ‘(2)

(cons (cons 1 ‘()) (cons 2 ‘())) => ‘((1) 2)

Test this (and try additional examples) in DrRacket

More About cons

A cons cell contains two fields

- first [also called car]
- rest [also called cdr]

For a list the rest field must be a list

Generally both fields of a cons $\in$ All

- (cons 1 2) = (1 . 2)
- Called a dotted pair
- first and rest require that the inputs are lists whereas car and cdr do not

Recursive Definition

- Recursive Definition

  List : empty | cons All List

  Process lists using these two cases and use recursion to recursively process lists in cons
- Use null? to check for empty list and cons? to check for a cons
- Use first and rest to access components of cons

List Predicate

- Follow the definition

```
(define (list? L)
 (if (null? L)
     #t
     (if (cons? L)
         (list? (cdr L)) ; need cdr not rest since L arbitrary
         #f)))
```

Version with cond

```
(define (list? L)
 (cond
    [(null? L) #t]
    [(cons? L) (list? (cdr L))]
    [else #f]))
```

Non-Terminating

- Make sure recursive calls are smaller

```
(define (list? L)
 (if (null? L)
     #t
     (if (cons? L)
         (list? L)
         #f)))
```

Length[1]

```
(define (length L)
  (if (null? L)
    0
   (                              ) ))
```

1: length is a built-in function in Racket

Length[1]

```
(define (length L)
  (if (null? L)
    0
   (+ 1 (length (rest L))) ))
```

- The recursive function can be "thought of" as a definition of length

1: length is a built-in function in Racket

Member[1]

```
(define (member x L)
  (cond
   [(null? L) #f]
   [                   ]
   [                   ]))
```

1. Member is a built-in function in Racket and the specification is different

Member[1]

```
(define (member x L)
  (cond
   [(null? L) #f]
   [(equal? x (first L)) #t]
   [else (member x (rest L))]))
```

1. Member is a built-in function in Racket and the specification is different. This generalizes the function from Chapter 2 of the Little Schemer which works on atoms and uses eq?

Append[1]

```
(define (append x y)
…
)
```

- (append ‘(1 2 3) ‘(4 5 6)) $\rightarrow$ (1 2 3 4 5 6)

- Recurse on the first input

1. append is a built-in function in Racket

Append[1]

```
(define (append x y)
  (if (null? x)
    y
    (cons (first x) (append (rest x) y))))
```

- Recurse on the first input

1. append is a built-in function in Racket

Reverse[1]

```
(define (reverse L)
…
  )
```

- (reverse ‘(1 2 3)) $\rightarrow$ (3 2 1)

1. reverse is a built-in function in Racket

Reverse[1]

```
(define (reverse L)
  (if (null? L)
    null
    (append (reverse (rest L)) (cons (first L) null))))
```

1. reverse is a built-in function in Racket

Exercise

Implement the function (snoc x L) which returns the list obtained from L by inserting x at the end of L.

1) Use reverse
2) Use recursion similar to reverse

snoc

```
(define (snoc x L)
  (reverse (cons x (reverse L)))

(define (snoc x L)
  (cond
    [(null? L) (cons x null) ]
    [else (cons (first L) (snoc x (rest L)))]
    ))
```

Tail Recursive reverse

```
(define (reverse-aux x y)
  (if (null? x)
    y
    (reverse-aux (rest x) (cons (first x) y))))

(define (reverse x)
   (reverse-aux x null))
```

- $O(n)$ vs $O(n^2)$

Numatoms

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))

; return number of non-null atoms in x
(define (numatoms x)
  (cond
   [(null? x) 0]
   [(atom? x) 1]
   [(cons? x) (+ (numatoms (first x))
              (numatoms (rest x)))]))
```

Shallow vs Deep Recursion

- Length and Member only recurse on the rest field for lists that are conses
  - Such recursion is called shallow – it does not matter whether the lists contain atoms or lists
  - (length ‘((a b) c d)) => 3
- Numatoms recurses in both the first and rest fields
  - Such recursion is called deep – it completely traverses the list when elements are lists
  - (numatoms ‘((a b) c d)) => 4

Termination

- Recursive list processing functions, whether shallow or deep must eventually reach the base case.
- The inputs to each recursive call must have a smaller size
  - Size is the number of cons cells in the list
  - (< (size (first l)) (size l))
  - (< (size (rest l)) (size l))

Size

```
(define (atom? x)
  (not (cons? x)))
; return size of x = number of cons cells in x
(define (size x)
  (cond
   [(atom? x) 0]
   [(null? x) 0]
   [(cons? x)  (+ 1 (size (first x)) (size (rest x)))]))
```

Order

```
; return the order = maximum depth
(define (order x)
  (cond
   [(null? x) 0]
   [(atom? x) 0]
   [(cons? x)  (max (+ 1 (order (first x)))
                    (order (rest x)))]))
```

Higher Order Functions

sort:

```
> (sort '(4 3 2 1) <) => (1 2 3 4)
> (sort '("one" "two" "three" "four") string<?) =>
'("four" "one" "three" "two")
```

map:

```
>  (map sqr '(1 2 3 4)) => ‘(1 4 9 16)
```

Higher Order Functions

filter:

```
>    (filter odd? '(1 2 3 4 5)) => ‘(1 3 5)
>    (filter even? ‘(1 2 3 4 5)) => ‘(2 4)
```

fold:

```
>   (foldr cons '() '(1 2 3 4)) => ‘(1 2 3 4)
>   (foldr list '() '(1 2 3 4)) => '(1 (2 (3 (4 ()))))
>   (foldr + 0 '(1 2 3 4)) => 10
>   (foldl cons ‘() ‘(1 2 3 4)) => ‘(4 3 2 1)
>   (foldl list '() '(1 2 3 4)) => '(4 (3 (2 (1 ()))))
>   (foldl * 1 ‘(1 2 3 4)) => 24
```

filterodd

```
; Input: a list L
; Output: a list with the odd elements removed
(define (filterodd L)
  (cond
    [(null? L) L]
    [(odd? (first L)) (cons (first L) (filterodd (rest L)))]
    [else (filterodd (rest L))]))
```

filtereven

```
; Input: a list L
; Output: a list with the even elements removed
(define (filtereven L)
  (cond
    [(null? L) L]
    [(even? (first L)) (cons (first L) (filtereven (rest L)))]
    [else (filtereven (rest L))]))
```

filter

```
; Input: a list L and a predicate p?
; Output: a list with the elements satisfying p? removed
(define (filter p? L)
  (cond
    [(null? L) L]
    [(p? (first L)) (cons (first L) (filter p? (rest L)))]
    [else (filter p? (rest L))]))
```

Order Using map/reduce

```
; return the order = maximum depth
(define (order x)
  (cond
   [(null? x) 0]
   [(atom? x) 0]
   [(cons? x) (+ 1 (foldr max 0 (map order x)))]))
```

Flatten Using map/reduce

```
; return a first order list containing all atoms in x
(define (flatten x)
  (cond
    [(null? x) x]
    [(atom? x) (list x)]
    [(cons? x) (foldr append '() (map flatten x))]))
```