Lists & Functional Recursion

# General Information on Lists

|  |
| --- |
| **box-and-pointer diagram #1** |
| listOverallView  Important!!  cdr  car |

* think of it as ***series of expressions*** enclosed in parentheses
* elements (or atoms)
  + **are immutable**
  + has a (pointer to) a value (car of list) and a pointer to the next node (cdr of list)
  + last node ‘s cdr pointer is to null
  + Holds ONLY ONE value per element!!
  + but an element can be a list, function, or expression itself!!!

|  |
| --- |
| **box-and-pointer diagram #1** |
| listOverallView |

* overall structure
  + **is mutable** (but the data inside is not)
  + Heterogeneous
    - elements DO NOT have to be of the same data type
  + is recursive structures (linked lists)
  + not accessible by index number

|  |
| --- |
| **Example Lists Code** |
| ;creates a list (side effect)  (list 2 3 5 7)  (list 'House 'at 'Pooh 'Corner)  (list 4 'calling 'birds 3 'French 'hens 2 'turtle 'doves 1)  (list 'buckle 'my 'shoe)  (list 'one (list 2 (list 'three 4)) 5 'six)  (list 'how (list 'nested (list 'can (list 'u (list 'go)))) 'how (list 'nested (list 'can (list 'u (list 'go)))))  (list 23 876 'Lupoli 23 'Stinks) ; list build function  ;creates a temp list called a literal list  '(1 2 3 4 5)  '() ; used often to compare if an empty list  ; ‘ is used to tell Scheme NOT to eval the first item inside ( )  (list 'Yankees 2 'Diamondbacks 1) ; pictured ***above***  (list 'A (list 'B 'C) 'D (list 'E (list 'F 'G))) ; pictured below |

|  |
| --- |
| **box-and-pointer diagram #2** |
| Picture 2 |

|  |
| --- |
| **box-and-pointer diagram #3** |
| basicListCreationEx1  (less detailed, but notice the overall structure of the separate lists) |

# Basic Operators on a Scheme List

* car
  + first item (element/atom) in the node
  + accesses ***individual element***
    - but that element COULD BE EITHER an atom or a start of another inner list!!
* cdr
  + “coo-der”
  + all items AFTER the ***current*** item in the list
  + super helpful with recursion
  + accesses ***rest of list***
* cons
  + constructs a new list, beginning with its first argument and continuing with its second
  + more on this later
* origins of car, cdr, and con
  + names date back to 1958
  + Before lower case characters were invented
    - CONS = CONStruct
    - CAR and CDR were each implemented by a single hardware instruction on the IBM 704
      * CAR: Contents of Address Register
      * CDR: Contents of Data Register

# Working & Combining CDR/CAR with a flat list

* simple, think of it as a flat linked list
  + no sub lists
* can combine car and cdr
  + (car (cdr (cdr ‘(1 2 3 4 5)))) == (caddr ‘(1 2 3 4 5))
* each node has a link to another node, except last one
* car/cdr is simple, returns respect portion of flat list
* does this in a STRAIGHT linear line
  + this becomes important when inner lists are introduced
* **notice it appears the latter portion of the command is done first!!**
  + **caddr – travels d, then d, then a!!**

|  |  |
| --- | --- |
| Working with a Flat Scheme List | |
| C:\Users\Luper\Desktop\SchemeFlatListExercises.png | |
| (car '(A B C D E F))  (cdr '(A B C D E F)) | A  (B C D E F) |
|  | |
| Prove these correct ***and stop.***  (cadr '(A B C D E F))  (cddr '(A B C D E F))  (caddr '(A B C D E F))  (cdddr '(A B C D E F)) | B  (C D E F)  C  (D E F) |

What would this return?? Draw it out!!

1. (car (cdr (cdr ‘(1 2 3 4 5)))) => ??? Answer: 3
2. (cddr ‘(1 2 3 4 5)) => ??? Answer: (3 4 5)
3. Get into (compute/linux), use GSI in interpretative mode and determine if your answers are correct.
4. What would the code be to get ***the element*** 5? Answer:

(car (cddddr ‘(1 2 3 4 5)))

(car ( cdr (cdddr ‘(1 2 3 4 5))))

(caddddr ‘(1 2 3 4 5)) // error

# What does (cdddddr ‘(1 2 3 4 5 6 7 8 9)) return? But why did #2 work? Answer:

Sadly, there is a limit to concatenated car/cdrs in GSI Scheme (max of 4)

Working with CDR/CAR in a complex list

* car/cdr **still** work in a ***linear*** fashion
  + each “d” in cdr traverses the TOP layer of the linked list
  + but it will return any inner lists it finds
* notice the code on how to create a complex list with inner lists

|  |
| --- |
| Layers of a Complex List |
| C:\Users\Luper\Desktop\SchemeTopLayerOfComplexList.png |

* car/cdrs still works the same, BUT cdr will return and travel (in order) the sublists, then back OUT to the top layer
  + this is also recursive!

|  |
| --- |
| **Interacting with Inner Lists** |
| basicListCreationEx1 |

|  |  |
| --- | --- |
| Code | Response |
| (print (cdr '(A (B C) D (E (F G))))) | ((B C) D (E (F G))) |
| (print (cddr '(A (B C) D (E (F G))))) | (D (E (F G))) |
| (print (cdddr '(A (B C) D (E (F G))))) | ((E (F G))) |
| (print (cddddr '(A (B C) D (E (F G))))) | () |
| (one more on next page) |  |
| (print (cdddddr '(A (B C) D (E (F G))))) | reference to undefined identifier: cdddddr |

|  |
| --- |
| **Interacting with Inner Lists Exercise** |
| 1. What will (print (cadddr ‘(A (B C) D (E (F G))) ) return? Answerb:   basicListCreationEx1  Answer: (E (F G))   1. What could    1. (print (caddddr ‘(A B C D E (F G))) return theoretically?    2. (draw it out)    3. But using GSI Scheme, how would I code it? |

# Cons

* way of ***adding to the front of the list***!!
* the con “node” acts like a linked list with a few “interesting” features
  + node becomes a pointer **(x 2)** to two ***items*** (could be a list or atom)
  + one ptr points to whatever was in arg[1] in the statement
  + one ptr points to whatever was in arg[2] in the statement
    - remember arg[0] ***is*** cons
* notice the minor (but important) difference of adding either the atom or another list
  + “cons” pointer remains the same
  + notice results (if inserting a list), now contains an inner list
  + ( ), or empty list is a legit value

|  |  |  |
| --- | --- | --- |
| **As evaluated (and pictured)** | | |
| Scheme code | Resulting Structure | Resulting Response |
| (cons 1 ‘(2 3))  **// an atom** |  | (1 2 3) |
| (cons ‘(1 2 3) ‘(4 5))  **// a list** | ( (1 2 3) 4 5) |
| (cons ‘A ‘()) | (A) |
| (cons '(A B) '(C D)) | ((A B) C D) |
| (cons '() '(A B)) | (() A B) |
| (cons 'A '(B C)) | (A B C) |

# multiple cons

# is useful in linking two lists together, but notice the results

|  |
| --- |
| Multiple Con Example |

|  |  |
| --- | --- |
| Code | Result |
| (cons 1 ‘(2 3))(cons (cons 1 2) (cons 3 4)) | (1 2 3) ((1 . 2) 3 . 4) |

# Draw what these would look like. Remember, what am I connecting, an atom or a list? Answerb:Other useful list functions

* predicate functions for list data
  + (null? S) tests if S is the empty list “( )”
    - ***used a lot*** in recursive functions
  + (list? S) tests if S is a list
  + (equal? L1 L2)
    - entire lists
  + (eq? element1 element2)

|  |
| --- |
| List Predicate Functions |
| (null? ‘(1 2 3) => #f  (null? ‘()) => #t  (list? ‘(1 2 3)) =>#t  (list? ‘3) => #f |

* list functions
  + list - no ? - makes a list of its arguments
  + “append” concatenates two lists
    - append takes any number of arguments
    - Append constructs ***new*** list
  + (length L)
    - evaluates (returns) the length of list L
    - “length” is the number of top-level elements in the list

|  |  |
| --- | --- |
| **List Functions Examples** | |
| (list 'A '(B C) 'D) => (A (B C) D)  (list (cdr '(A B)) 'C) => ((B) C)  (list ‘A) => (A)  (list) => ( )  (length (list 'A '(B C) 'D) ) => 3 Why??  (length (list )) => 0 | (append ‘(1 2) ‘(3 4)) => (1 2 3 4)  (append '(A B) '((X) Y)) => (A B (X) Y)  (append ‘( ) ‘(1 2 3)) => (1 2 3)  (append ‘(1) ‘(2 3) ‘(4 5 6)) => (1 2 3 4 5 6)  (append ‘(1 2)) => (1 2)  (append) => null  (append null null null) => null |

* Overall notes
  + Note that the parenthesized prefix notation makes it easy to define functions that take a varying number or arguments

# Binding variable names with Lists

# whenever we define a function, list, etc… a pointer is used

# there is a chance of have two (or more pointers) pointing to the same value

# this is where eq? (same object) and equal? (same data) differ

|  |  |
| --- | --- |
| **Scheme and Pointers** | |
| Example 1 | Example 2 |
|  |  |

# Draw what the various lists would look like:

|  |  |
| --- | --- |
| Code | Draw Box and Pointer Diagram Here |
| 1.  (define x '(a))  (define z4 (list x x)) |  |
| 2.  (define x '(a))  (define y (cons x x)) |  |
| 3.  (define zi '(a b c)) |  |
| 4.  (define Z (list ‘a (list ‘b ‘c) ‘d)) |  |

Answerb:

Mathematical Recursion

* just a refresher
* in Scheme, the data will be lists

Work on these on another sheet of paper

|  |
| --- |
| **f(10) = ??? (I’ll do this one)** |
| |  |  |  | | --- | --- | --- | | **f(x) =** | { | **f(x-3)+2 x>1** | | **x + 3** | |
| |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | | **F(10)** | **=** | **F(7)+2** | | **“call” stack** |  | **“value” stack** | |
| **f(9) = ???** |
| |  |  |  | | --- | --- | --- | | **f(x) =** | { | **2 \* f(x-3) x>= 2** | | **5** | |
| **f(9) = ???** |
| |  |  |  | | --- | --- | --- | | **f(x) =** | { | **3** | | **2\*f(x-2) x >= 2** | |  |  |  | |
| **f(7) = ???** |
| |  |  |  | | --- | --- | --- | | **f(x) =** | { | **f(x-3)+2 x>1** | | **x + 3** | |
| **F(10) = ??? How about f(5)???** |

|  |
| --- |
| **Tail Recursion – Fibonacci** |
| (define (fib n)  (if (< n 2) n  (+ (fib (- n 1)) (fib (- n 2)))  )  )  (fib 10) => 55 |

# Car/Cdr Recursion (List recursion)

* using car/cdr we pass a different piece of the list for the same function to evaluate!!
  + car = access INDIVIDUAL element (or atom) data
  + cdr = the REST of the list
* usually, anything we want to repeat, we use recursion, but pass the REST of the list using CDR
* ***At runtime*** type mismatches are detected!!!!
* So minimal code needed for datatypes.
  + Behind the scenes it is only breaking things into tokens, THEN when interpreting does it do type check

(+ “Lupoli” 3.14159 ‘(1 2) ) ; the + will fail at the string

# Using Recursion to Find an item in a List

* again using car/cdr is invaluable
  + cdr along with the recursive call enables us to check the NEXT atom in the list
* base case is always looking for an empty list
  + since we may have gone through the entire list!

|  |
| --- |
| **Tail Recursion - Member** |
| ; member takes an atom and a simple list;  ; returns #t if the atom is in the list; #f otherwise  (define (member atm lis)  (cond  ((null? lis) #f) **#1**  ((eq? atm (car lis)) #t) **#2**  (else (member atm (cdr lis))) **#3**  )  )  (member 2 '(1 2 3 4)) => #t  (member 5 '(1 2 3 4)) => #f  (member "B" '("A" "H" "B" "D" "W" "E" "R")) => #t |

|  |
| --- |
| **Tail Recursion - Member** |
| ; member takes an atom and a simple list;  ; returns #t if the atom is in the list; #f otherwise  (define (member atm lis)  (cond  ((null? lis) #f) ; returns, **no** calls **#1**  ((eq? atm (car lis)) #t) ; returns, **no** calls **#2**  (else (member atm (cdr lis))) **#3**  )  )  (member 2 '(1 2 3 4)) => #t  ***Please notice that the else clause is the ONLY portion that calls another function!!*** |

|  |  |
| --- | --- |
| Watching the List Recursion | |
| |  | | --- | |  | |  | | (member 2 '(2 3 4)) | | (member 2 '(1 2 3 4)) | | Call Stack | | |  | | --- | |  | |  | | **#2** ((eq? atm (car lis)) #t | | **#3** (member 2 '(~~1~~ 2 3 4)) | | Value Stack | |

|  |
| --- |
| **Looking Through a List Recursively** |
| ; member takes an atom and a simple list;  ; returns #t if the atom is in the list; #f otherwise  (define (member atm lis)  (cond  ((null? lis) #f) **#1**  ((eq? atm (car lis)) #t) **#2**  (else (member atm (cdr lis))) **#3**  )  )  (member 5 '(1 2 3 4)) => #f  (member "B" '("A" "H" "B" "D" "W" "E" "R")) => #t |

|  |  |
| --- | --- |
| Watching the List Recursion | |
| |  | | --- | |  | | (member 5 '()) | | (member 5 '(4)) | | (member 5 '(3 4)) | | (member 5 '(2 3 4)) | | (member 5 '(1 2 3 4)) | | Call Stack | | |  | | --- | |  | | **#1** ((null? lis) #f) | | **#3** (member 5 '(~~4~~)) | | **#3** (member 5 '(~~3~~ 4)) | | **#3** (member 5 '(~~2~~ 3 4)) | | **#3** (member 5 '(~~1~~ 2 3 4)) | | Value Stack | |

# Using Recursion to Accumulate data in a List

* again using car/cdr is invalueable
  + cdr along with the recursive call enables us to check the NEXT atom in the list
* base case is always looking for an empty list
  + can “return” a nominal value that won’t effect the data
    - adding 0 is a value for total
    - or return nothing!!
* many of the calls have to wait for the OTHERS in the stack to finish

|  |
| --- |
| **Accumulating Data in the List Recursively** |
| (define (list-sum lst)  (cond  ((null? lst) 0) ; returns, **no** calls **#1**  (else (+ (car lst) (list-sum (cdr lst)))) **#2**  )  )  ; Notice how 0 is “returned” |

|  |  |  |
| --- | --- | --- |
| Watching the List Recursion | | |
| |  | | --- | |  | | (list-sum '())) | | (list-sum '(7))) | | (list-sum '(6 7))) | | (list-sum '(3 6 7))) | | (list-sum '(12 3 6 7))) | | Call Stack | | |  | | --- | |  | | **#1** ((null? lis) 0) | | **#2** (+ (7)(list-sum (~~7~~))) | | **#2** (+ (6)(list-sum (~~6~~ 7))) | | **#2** (+ (3)(list-sum (~~3~~ 6 7))) | | **#2** (+ (12)(list-sum (~~12~~ 3 6 7))) | | Value Stack | | |  | | --- | |  | | 0 | | 7 + 0 | | 6 + 7 | | 3 + 13 | | 12 + 16 | | 28!! | |

|  |
| --- |
| **Accumulating Data with List Recursively** |
| (define (list-sum lst)  (cond  ((null? lst) 0) **#1**  (else (+ (car lst) (list-sum (cdr lst)))) **#2**  )  ) |

Complete the Stacks for the call below:

|  |  |
| --- | --- |
| Watching the List Recursion - Exercise | |
| |  | | --- | |  | | (list-sum ‘()) | | (list-sum ‘(3)) | | (list-sum ‘(2 3)) | | (list-sum ‘(1 2 3)) | | (list-sum ‘(“Lupoli” 1 2 3)) | | Call Stack | | |  | | --- | |  | | **#1** (null? lst) 0 | | **#2** (+ (3)(list-sum ())) | | **#2** (+ (2)(list-sum (3))) | | **#2** (+ (1)(list-sum (2 3))) | | **#2** (+ (“Lupoli”)(list-sum (1 2 3))) | | Value Stack | |

Answerb:

# Using Recursion to Append data to a List

* VERY SIMILAR to accumulating
* again using car/cdr is invalueable
  + cdr along with the recursive call enables us to ***add*** the NEXT atom in the list
* adding the first parameter list, to the second parameter list (base)
* base case is always looking for an empty list
  + can “return” the rest of the list (second parameter)
* many of the calls have to wait for the OTHERS in the stack to finish to complete the entire list

|  |
| --- |
| **Advanced Append Function #1** |
| (define (append l1 l2)  (if (null? l1) l2 ; if true, and l1 is null **#1**  (cons (car l1) (append (cdr l1) l2))))) **#2**  ; combine with cons  (print (append '(X Y Z) '(A b c))) => (X Y Z A b c) |

|  |
| --- |
| Watching the List Recursion |
| Call Stack |
| |  |  | | --- | --- | | Step |  | | #4 | (append'() '(A b c)) | | #3 | (append'(Z) '(A b c)) | | #2 | (append'(Y Z) '(A b c)) | | #1 | (append'(X Y Z) '(A b c)) | |  | Call Stack | |

|  |
| --- |
| Value Stack Step #1 |
| (cons (car l1) (append (cdr l1) l2))) |
| Value Stack Step #2 |
| (cons (car l1) (append (cdr l1) l2))) |

|  |
| --- |
| Value Stack Step #3 |
| (cons (car l1) (append (cdr l1) l2))) |
| Value Stack Step #4 |
| (null? l1) l2 |

Scheme Example Code(s)

* Show different examples for different goals
* Please notice how the parameters are set up
  + parameter that starts with
    - l? = list (ex. l2)
    - a? = atom (single element) (ex. atm)
* Please notice HOW they are completing the goal required
  + some pare passing **car**, most are passing **cdr**
  + in appending, notice it CONS the car with the rest of the list

|  |
| --- |
| **Tail Recursion - equalLists** |
| ; equalList takes two general lists as parameters  ; returns #t if the two lists are equal  ; #f otherwise  **; Scheme already has this function – equal?**  (define (equalLists l1 l2)  (cond  ((not (lt? l1))(eq? l1 l2))  ((not (lt? l2)) #f)  ((null? l1) (null? l2))  ((null? l2) #f)  ((equalLts (car l1) (car l2))  (equalLts (cdr l1) (cdr l2)))  (else #f)  )  )  (equalLists '(1 2 3 4) '(1 2 3 4)) => #t  (equalLists '(1 2 3 4) '(1 2 4)) => #f |

|  |
| --- |
| **Advanced Append Function #2** |
| (define (append l1 l2)  (cond ((null? l1) l2)  ((cons (car l1) (append (cdr l1) l2))))) |

|  |
| --- |
| **Sorted? Example** |
| ; checks to see if the **numbered** list is sorted  ; accepts integer list, return true if sorted  (define (sorted? numL)  (cond  ((<= (length numL) 1) #t)  ((<= (car numL) (cadr numL)) (sorted? (cdr numL)))  (else #f)  )  )  (sorted? '( 1 2 2 4 7)) => #t  (sorted? '( 1 0 2 4 7)) => #f  (sorted? '( 7 8 9 1)) => #f |

Functional Recursion Steps for Success

* setup up your base case first
  + if you are finding something in a list, check if empty and return true/false
  + if you need to go through the list no matter what, you must return some nominal value
* if not the base case, but not done, always pass the REST of the list
  + else (member atm (cdr lis))
* remember how to “return” in a conditional statement
  + (<= (length numList) 1) #t)
  + (else #f)
* remember how to add elements to a list using cons

Built-in “primitive functions” for Lists

* all do exactly what you expect
* a complete list is below:
  + download.plt-scheme.org/doc/html/reference/numbers.html
* short list
  + min/max
  + lcd
  + gcd
  + filter (requires a function as a parameter)
    - the filter accepts a function (telling what to filter) as a parameter
  + etc…

|  |
| --- |
| Min Scheme Function |
| |  | | --- | | ([min](http://download.plt-scheme.org/doc/html/reference/numbers.html?q=min&q=filter&q=min" \l "%28def._%28%28quote._%7E23%7E25kernel%29._min%29%29) x ...+) → [real?](http://download.plt-scheme.org/doc/html/reference/numbers.html?q=min&q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._real%7E3f%29%29) | | x : [real?](http://download.plt-scheme.org/doc/html/reference/numbers.html?q=min&q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._real%7E3f%29%29) |   Returns the smallest of the xs. |
| |  | | --- | | > ([min](http://download.plt-scheme.org/doc/html/reference/numbers.html?q=min&q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._min%29%29) 1 3 2) | | 1 | | > ([min](http://download.plt-scheme.org/doc/html/reference/numbers.html?q=min&q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._min%29%29) 1 3 2.0) | | 1.0 | |

|  |
| --- |
| Filter Scheme Method |
| |  | | --- | | ([filter](http://download.plt-scheme.org/doc/html/reference/pairs.html?q=filter&q=min" \l "%28def._%28%28lib._scheme/base..ss%29._filter%29%29) pred lst) → [list?](http://download.plt-scheme.org/doc/html/reference/pairs.html?q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._list%7E3f%29%29) | | pred : [procedure?](http://download.plt-scheme.org/doc/html/reference/procedures.html?q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._procedure%7E3f%29%29) | | lst : [list?](http://download.plt-scheme.org/doc/html/reference/pairs.html?q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._list%7E3f%29%29) |   Returns a list with the elements of lst for which pred produces a true value. The pred procedure is applied to each element from first to last. |
| |  | | --- | | > ([filter](http://download.plt-scheme.org/doc/html/reference/pairs.html?q=filter&q=min#%28def._%28%28lib._scheme/base..ss%29._filter%29%29) [positive?](http://download.plt-scheme.org/doc/html/reference/numbers.html?q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._positive%7E3f%29%29) '(1 -2 3 4 -5)) | | (1 3 4) | |

Mapping within Lists

* map means apply a given rule (or function) to all data within the given list

‘(1 2 3 4) => E(double\_all) => ‘(2 4 6 8)

‘(1 2 3 4) => E(increment by +1) => ‘(2 3 4 5)

* an easy way to do this is to pass BOTH
  + function needed to map
  + data

|  |
| --- |
| **Map Example – Function Passing** |
| MzScheme |
| (define (double x) ( \* x 2)) ; quick, one line function  (define (increment x) ( + x 1)) ;  **; here we pass in a function into the parameter “func” in**  **; order to tell what mapcar needs to do**  (define (mapcar func list)  (if (null? list) '()  (cons (func (car list)) (mapcar func (cdr list)))))  (mapcar double ‘(1 2 3 4)) ;double HAS to be defined already!!  (mapcar increment ‘(1 2 3 4)) |
| GSI Scheme |
| **; same as above, just “map” instead of “mapcar”**  (define (map func list)  (if (null? list) '()  (cons (func (car list)) (map func (cdr list)))))  (map double ‘(1 2 3 4)) ;double HAS to be defined already!!  (map increment ‘(1 2 3 4)) |

# Flatten (Difference between append/con)

* return a complete list given various and numerous list
  + kinda like removing all of the inner ( )s
* uses recursion
* code for this is very clean and quick
* would require more code in C/C++
  + worry about nodes in Linked List
  + then memory management!!
  + all of this is taken care of for you in Scheme
* Scenarios
  + ‘() already flattened, just return
  + ‘( 1 ……) cons car (atom, not list), recursive flatten the cdr
  + ‘( (1 2) …..) append the flattening, recursive flatten the cdr

|  |
| --- |
| **Flattening a List** |
| (define (flatten list)  (cond ((null? list) '())  ((list? (car list)) (append **(**flatten (car list)**)** **(**flatten (cdr list)**)**))  (else (cons (car list) (flatten (cdr list))))  )  )  (flatten ‘(1 2 3 4)) => 1 2 3 4  (flatten ‘(1 (2 “3”) 4 ((5)))) => 1 2 “3” 4 5  difference between cons and append?? |

# Symbols (Mathematical) As Parameters

* remember we can pass functions as parameters
* operators (<, >, ==, !=, etc…) are really functions too

|  |
| --- |
| **A more capable sort? function** |
| ; checks to see if the list is sorted  ; accepts integer/String list, return true if sorted  (define (sorted? numList comparator)  (cond  ((<= (length numList) 1) #t)  ((comparator (car numList) (cadr numList)) (sorted? (cdr numList) comparator))  (else #f)  )  )  (sorted? '(1 2 2 4 7) <= ) => #t  (sorted? '("a" "b" "f" "c") string<? ) => #f  (sorted? '(9 8 7 1) > ) => #t |

On your own:

1. Create a function “average” that given an item, will determine the average value in the list. The function will need to be recursive.
2. Given a list of numbers, increase the list’s value by 10%. Use mapping.
3. Given a list of numbers, count the negative values
4. Reverse a give list.

# Theory of List Comprehension

* lists are one of the most used and basic data structure for holding values for computation
* powerful ***comprehension*** notation/syntax to manipulate these lists
  + syntactic sugar – easy concise code to create something powerful
  + simply put… great and simple syntax to create and manipulate lists
    - don’t need ***create*** explicit recursion to traverse/edit a list
      * we have functions to do that for us!!
* there are many languages that have this feature
  + <https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(list_comprehension)#Scheme>
  + but ***not*** Scheme (unless an additional library is added)
    - but it does have concise functions and syntax to do the same things

Please take a look at Haskell, Java and Java Script from the link above.

Also notice which popular languages are NOT on the list!

# Resources:

<http://theopenacademy.com/sites/default/files/oadb/Sciences/Computer%20Science/Computer%20Science%20III;%20Programming%20Paradigms%20-%20Stanford%20-%20Jerry%20Cain%20-%20BYNCSA/Assignments/Problem%20Sets/Problem%20Set%201.pdf>

<http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-22.html#%_fig_3.12>

<http://theopenacademy.com/content/lecture-20-car-cdr-recursion-problem>

download.plt-scheme.org/doc/html/reference/numbers.html

List Comprehensions

<https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(list_comprehension)#Scheme>

# FYI Section

# Figuring out Errors

|  |
| --- |
| **Finding the line number with the error??** |
| procedure application: expected procedure, given: (); arguments were: (X Y Z)  === context ===  /afs/umbc.edu/users/s/l/slupoli/home/CMSC331/code/Scheme/lists/appendExample.scm:1:0: advAppend  Error was on line #1 |

Answers

|  |
| --- |
| Ex. #1 - Using cddr on a flat list |
|  |
|  |

|  |
| --- |
| Ex. #1 Inner Lists - cadddr on a complex list |
|  |

|  |
| --- |
| Ex. #2 Inner Lists - Limits to cdddr characters |
| ( cadr ( cdddr ‘(A B C D E (F G)) ) ) |

|  |
| --- |
| Multiple Cons (b) |
|  |

|  |  |
| --- | --- |
| List Binding | |
| 1.  (define x '(a))  (define z4 (list x x)) |  |
| 2.  (define x '(a))  (define y (cons x x)) |
| 3.  (define zi '(a b c)) |
| 4.  (define Z (list ‘a (list ‘b ‘c) ‘d)) |

|  |
| --- |
| **Accumulating Data List Recursively** |
| (define (list-sum lst)  (cond  ((null? lst) 0) **#1**  (else (+ (car lst) (list-sum (cdr lst)))) **#2**  )  ) |

|  |  |
| --- | --- |
| Watching the List Recursion - Exercise | |
| |  | | --- | |  | | (list-sum ‘()) | | (list-sum ‘(3)) | | (list-sum ‘(2 3)) | | (list-sum ‘(1 2 3)) | | (list-sum ‘( “Lupoli” 1 2 3)) | | Call Stack | | |  | | --- | |  | | **#1** (null? lst) 0 | | **#2** (+ (3)(list-sum ())) | | **#2** (+ (2)(list-sum (3))) | | **#2** (+ (1)(list-sum (2 3))) | | **#2** (+ (“Lupoli”)(list-sum (1 2 3))) | | Value Stack | |

* Thanks Mike Jackson (Su12)

|  |
| --- |
| **is?** |
| DEFINE (is? atm lis)  (COND  ((NULL? lis) #F)  ((EQ? atm (CAR lis)) #T)  ((ELSE (is? atm (CDR lis)))  )) |

|  |
| --- |
| **equalList?** |
| (DEFINE (equalsimp lis1 lis2)  (COND  ((NULL? lis1) (NULL? lis2))  ((NULL? lis2) #F)  ((EQ? (CAR lis1) (CAR lis2))  (equalsimp(CDR lis1)(CDR lis2)))  (ELSE #F)  )) |

|  |
| --- |
| **Determining the average of a list (using apply)** |
| (define (average aList)   (cond     ((null? aList) 0)     ((not (list? aList)) aList)     (else (/ (apply + aList) (length aList))))) |
| **Determining the average of a list** |
| (define (average2 aList)   (cond     ((null? aList) 0)     ((not (list? aList)) aList)     (else (/ (+ (car aList) (\* (average2 (cdr aList)) (length (cdr aList))))               (length aList))))) |

|  |
| --- |
| **Increasing the values in a list by 10** |
| manually mapping |
| (define (incrBy10 aList)   (cond     ((null? aList) '())     ((not (list? aList)) (if (number? aList) (\* 1.1 aList) aList))     (else (cons (incrBy10 (car aList)) (incrBy10 (cdr aList)))))) |
| but by mapping |
| (define (by10 x) ( + x 10)) (display (map by10 '(23 24 65 98))) |

|  |
| --- |
| **Count the negatives in a list (using Apply)** |
| (define (countNegative aList)   (cond     ((null? aList) 0)     ((not (list? aList)) (if (< aList 0) 1 0))     (else (apply + (cons (countNegative (car aList))                          (map countNegative (cdr aList))))))) |

|  |
| --- |
| **Reversing a List** |
| (define (reverseList aList)   (cond     ((null? aList) '())     ((not (list? aList)) (list aList))     (else (append (reverseList (cdr aList)) (list (car aList)))))) |

|  |
| --- |
| **Counting the negatives in a list (using lambda)** |
| (define (countNegative2 aList)   (length (filter (lambda (x) (< x 0)) aList))) |

Sources

List Basics

<http://www.cs.grinnell.edu/~rebelsky/courses/CS151/2007S/Readings/lists.html>