# toyCode

search

Home Scheme NASM Android

20th September 2014

## Set Difference

Question: Given two sets of integers A and B, write a Scheme program to find the difference A − B.

Set Difference A − B is defined as:

A − B = {x | x ∈ A and x /∈ B}

A and B are represented as lists.

Assume that none of the given lists contain duplicate elements.

Sample Input: (Set-Difference '(1 2 3 4) '(1 2))

Sample Output: (3 4)

```
(define check (lambda (a ls)
              (cond
              ((null? ls) #f)
              ((= a (car ls)) #t)
              (else (check a (cdr ls))))))
(define Set-Difference (lambda (ls1 ls2)
               (cond
               ((null? ls1) ls1)
               ((check (car ls1) ls2) (Set-Difference (cdr ls1)
ls2))
               (else (cons (car ls1) (Set-Difference (cdr ls1)
ls2))))))
```

Posted 20th September 2014 by Surya Harsha Nunnaguppala

Labels: foundations of programming, Functional Programming, Scheme, Set difference, Sets

0 Add a comment

18th September 2014

## Binary Tree: Family Tree : Descendent Tree

***Question:*** *Consider male members of a family with father-son relationship only, where at the maximum there are only two sons for a father. All the family members have unique names (you may assume that all have first names only). Person A is the ancestor of Person B if A is father / grand father / great grand father etc. of B. Person A is the descendent of person B if A is the son / grand* *son / great grand son etc. of B.Represent the family as a rooted binary tree; you must write all the necessary constructors and selectors.*

*1. Given a family and the name of a person, find the names of all his descendents.*
*2. Given a family and the name of a person, find the names of all his descendents who have no sons.*
*3. Given a family and the names of two persons, find their nearest common ancestor.*

```
(define (make-family node lb rb)     ;constructor, i/p-
node,left-branch,right-branch o/p-family-tree
             (list node lb rb))

(define (get-node family)                 ;selector, i/p-
family o/p-node/root
             (car family))

(define (get-lb family)                      ;selector, i/p-
family o/p-left-branch
        (cadr family))

(define (get-rb family)                     ;selector ,i/p-
family o/p-right branch
             (caddr family))

(define (mef)                                       ;makes a
empty family
             '())

(define (empty? family)                     ;i/p-family o/p-#t
for empty else #f
             (null? family))

(define (first a)                                     ;returns
1st of list
             (car a))

(define (rest a)                                     ;returns
list except first element
             (cdr a))

(define (main1 name family)             ;returns all the
descendants of person with mane "name" in family
              (cond ((empty? family) '())
                        ((eq? name (get-node family)) (append
(get-list (get-lb family))

                           (get-list (get-rb family))))
                       (else (append (main1 name (get-lb
family))
```

```
                                                (main1 name
(get-rb family))))))

(define (get-list family)                      ;returns list of
members in the family
             (cond ((empty? family) '())
                        (else (append (get-list (get-lb
family))
                                                (list (get-node
family))
                                               (get-list (get-
rb family))))))

(define (main2 name family)              ;returns all
descendants with no sons in the family of person with name
"name"
             (cond ((empty? family) '())
                       ((eq? name (get-node family)) (append
(get-list2 (get-lb family))

                         (get-list2 (get-rb family))))
                      (else (append (main2 name (get-lb
family)) (main2 name (get-rb family))))))


(define (get-list2 family)                   ;returns all the
descendants of family with no sons
             (cond ((empty? family) '())
                       ((and (empty? (get-lb family)) (empty?
(get-rb family)))
                       (list (get-node family)))
                       (else (append (get-list2 (get-lb
family)) (get-list2 (get-rb family))))))


(define (member-fam? name family) ;returns #t if person with
"name" exists in the family
             (mem? name (main1 (get-node family) family)))

(define (mem? name list-fam)           ;returns #t if person
with "name" exists in the family
            (cond ((empty? list-fam) #f)
                       ((eq? name (first list-fam)) #t)
                       (else (mem? name (rest list-fam)))))

(define (main3 n1 n2 family)                 ;returns common
ancestor of person1 and person2 with names n1 and n2
            (cond ((and (member-fam? n1 family) (member-fam? n2
family))
                             (cond ((and (member-fam? n1 (get-
lb family)) (member-fam? n2 (get-lb family)))
```

```
                                        (main3 n1 n2 (get-lb
family)))
                                       ((and (member-fam? n1
(get-rb family)) (member-fam? n2 (get-rb family)))
                                       (main3 n1 n2 (get-rb
family)))
                                       (else (get-node
family))))
                          (else 'invalid-names)))

;sample examples:

;(define a (make-family 'ab (make-family 'cd (mef) (mef))
(make-family 'ef (mef) (mef))))

;(define b (make-family 'gh (make-family 'ij (mef) (mef))
(make-family 'kl (mef) (mef))))

;(define c (make-family 'mn a b))

;(main1 'ab c) => o/p => (cd ef)

;(main2 'mn c) => o/p => (cd ef ij kl)

;(main3 'cd 'ef c) => o/p => ab
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: binary tree, descendent tree, Family tree, Functional Programming, Scheme, tree

0 Add a comment

## 18th September 2014 Procedural abstraction & data abstraction: Route log

***Question*** :There are many buses which run between Kochi and Kozhikode. Road Transport Office keeps the Trip-Record for each bus as registration-number of the bus, its starting-time at Kochi and the reaching-time in Kozhikode. The start time and end time each contain the hour and minute in 24 hour format. Route-Log for a day contains trip records for all the buses which run from Kochi to Kozhikode. Every day, the RTO would like to find out the bus which has taken the minimum time that day and warn its driver.
**Given a route log containing the trip records of Kochi-Kozhikode buses for a day, provide the design for obtaining the registration number of the fastest bus.**

Permitted assumptions:
• In one day, a bus makes only one trip from Kochi to Kozhikode.

- All buses start and end within the day (between 00:00 and 23:59). No midnight crossing journeys.

- There is no error checking needed for a trip record.

- The route log given is guaranteed to contain at least two trip-records.

Use of Procedural Abstraction and Data Abstraction are mandatory. All functions should be named appropriately and input-output specification stated precisely. Main function should be named as main. Constructors should start with *make*-and selectors with *get*-

```
;sample input : (main (list (make-triprec 1 (make-time 9 30)
(make-time 10 40))
;                            (make-triprec 2 (make-time 9 30)
(make-time 11 40))
;                             (make-triprec 3 (make-time 9 30)
(make-time 9 45))))
;o/p : 3 (the regno of the fastest bus)

(define (make-time h m)    ;i/p : hour and minutes,  o/p: list
of time/ time-rec
             (cons h (list m)))

(define (geth time)             ;i/p :time-rec , o/p: hour
             (car time))

(define (getm time)           ;i/p : time-rec, o/p :minutes
             (cadr time))

(define (make-triprec regno st et) ;i/p- bus-regno, start
time(st), end time(et), o/p:trip-record
             (cons regno (cons st (list et))))

(define (get-regno triprec)  ;i/p : trip-record, o/p: reg-no in
trip-record
             (car triprec))

(define (get-st triprec)         ;i/p: trip-record, o/p:start-
time
             (cadr triprec))

(define (get-et triprec)        ;i/p: trip-record, o/p:end-time
            (caddr triprec))

(define (get-time-taken rec)  ;i/p: trip-record, o/p: time-
taken
              (time-diff (get-st rec) (get-et rec)))

(define (time-diff st et)          ;i/p: start-time, end-time,
o/p:time-difference
             (if (>= (getm et) (getm st))
                  (make-time (- (geth et) (geth st))
                                     (- (getm et) (getm st)))
                  (make-time (- 1 (- (geth et) (geth st)))
                                     (- 60 (- (getm st) (getm
et))))))

(define (smallest rec1 rec2)      ;i/p/: two trip-records, o/p:
the trip-record of the fastest
             (cond ((< (geth (get-time-taken rec1)) (geth (get-
time-taken rec2))) rec1)
                        ((> (geth (get-time-taken rec1)) (geth
(get-time-taken rec2))) rec2)
                        ((< (getm (get-time-taken rec1)) (getm
(get-time-taken rec2))) rec1)
                        (else rec2)))

(define (get-first-rec rl)           ;i/p: routelog, o/p:first-
record of routelog
            (car rl))

(define (get-rest-rec rl)           ;i/p: routelog o/p:all
records except first record
             (cdr rl))

(define (empty? rl)                   ;i/p: routelog o/p: #t if
routelog is empty,else #f
             (null? rl))

(define (get-fastest rl)             ;i/p: routelog o/p:record
of the fastest
             (if  (empty? (get-rest-rec rl))
             (get-first-rec rl)
             (smallest (get-first-rec rl) (get-fastest (get-
rest-rec rl)))))

(define (main rl)                      ;i/p:routelog o/p:regno
of the fastest
             (get-regno (get-fastest rl)))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: bus, data abstraction, Functional Programming, procedural abstraction, route log, Scheme

0 Add a comment

18th September 2014

## Data Abstraction: Rational

## number implementation

***Question: (Data Abstraction example 2):***

*1) I want to represent a rational number p/q as ((p) (q)). Write make-rat, get-nr and get-dr accordingly.*
*2) Write functions plus-rat, mul-rat, print-rat using constructors and selectors of (1).*
*3) Implement equal-rat? using gcd (hcf), that is, q1 = q2 iff*

*(nr q1) x gcd (dr(q1), dr(q2)) / (dr q1) = (nr q2) x gcd (dr(q1), dr(q2)) / (dr q2)*
*4) Given a list of student-records where each record contains a roll no. and his/her marks, write a function to print the roll no.s of those students who have the same marks. All marks are rational numbers.Hint: This problem specification is ambiguous. What if two people have the same mark m1 and another three m2? Make your own decision and specify the problem properly before attempting to provide the solution.*

```
(define (make-rat p q) ;CONSTRUTOR : makes the rational no in
the given form
             (cons (list p) (list (list q))))

(define (get-nr ratno) ;SELECTOR: returns the numerator of
rational number ratrno
             (caar ratno))

(define (get-dr ratno) ;SELECTOR: returns the denominator of
the rational number ratno
            (caadr ratno))

(define (plus-rat r1 r2) ;returns r1+r2
             (make-rat (+ (* (get-nr r1) (get-dr r2))
                                (* (get-nr r2) (get-dr r1)))
                                (* (get-dr r1) (get-dr r2))))

(define (mul-rat r1 r2) ;returns the r1*r2
             (make-rat (* (get-nr r1) (get-nr r2))
                                (* (get-dr r1) (get-dr r2))))

(define (print-rat  ratno) ;to display the rational number
             (display (get-nr ratno))
             (display "/")
             (display (get-dr ratno)))

(define (equal-rat? r1 r2) ; returns #t if both rational
numbers are equal, else #f
             (if (= (/ (* (get-nr r1) (gcd (get-dr r1) (get-dr
r2)))
                          (get-dr r1))
                   (/ (* (get-nr r2) (gcd (get-dr r1) (get-dr
r2)))
                      (get-dr r2)))
            #t
            #f))

;;Implementation of the part 4 of question
(define (first-rec records) ; returns the first record of list
of records
             (car records))

(define (rest-rec records) ; returns structure of records
excluding the first record
             (cdr records))

(define (get-mark record) ;SELECTOR :returns the mark in the
record
             (cadr record))

(define (get-rno record) ;SELECTOR : returns the roll no of the
record
             (car record))

(define (equal-records? rec1 rec2)  ;returns #t when marks of
rec1 and rec2 are equal.
             (if (equal-rat2? (get-mark rec1) (get-mark rec2))
                  #t
                  #f))

(define (exists-dup? record records) ;returns #t if there
exists a record in i/p records which has same marks as that of
i/p record.
             (cond ((null? records) #f)
                        ((equal-records? record (first-rec
records)) #t)
                        (else (exists-dup? record (rest-rec
records)))))

(define (get-dup record records) ;returns a list rollno of
records from i/p records which have equal marks as that of i/p
record
             (cond ((null? records) '())
                        ((equal-records? record (first-rec
records)) (cons (get-rno (first-rec records)) (get-dup record
(rest-rec records))))
                        (else (get-dup record (rest-rec
records)))))

(define (remove record reclist) ; removes all those records
from reclist which have same marks as i/p record
             (cond ((null? reclist) '())
                       ((equal-records? record (first-rec
reclist)) (remove record (rest-rec reclist)))
                       (else (cons (first-rec reclist) (remove
record (rest-rec reclist))))))

(define (get-marks-same records) ; gives a list with sublists
of rollnos and each sublist is the list of people having same
marks
            (cond ((null? records) '())
                      ((exists-dup? (first-rec records) (rest-
rec records))
                      (cons (get-dup (first-rec records)
records) (get-marks-same (remove (first-rec records)
records))))
                      (else (get-marks-same (rest-rec
records)))))
```

*NOTE:*

*In the above example make the following changes :*

*1) I would like the representation to look like (p q). Write the constructor make-rat and selectors - get-nr and get-dr accordingly.*

*2) Implement equal-rat? which behaves the same way using another approach.*

*And also check out that the implementation of other functions are not changed even now and they work properly.*

*So the only changes are:*

```
(define (make-rat p q) ;CONSTRUTOR : makes the rational no in
the given form
             (cons  p (list q)))

(define (get-nr ratno) ;SELECTOR: returns the numerator of
rational number ratrno
             (car ratno))

(define (get-dr ratno) ;SELECTOR: returns the denominator of
the rational number ratno
            (cadr ratno))


(define (equal-rat2? r1 r2) ;another method to check equality
of rational numbers
        (if (= (* (get-nr r1) (get-dr r2))
                (* (get-nr r2) (get-dr r1)))
            #t
            #f))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: Functional Programming, Rational number, Scheme

0 Add a comment

18th September 2014

## Data Abstration: Find Youngest

***Question :(to be done using data abstraction):**Consider a collection (database) of persons denoted by person-db. Each person in this collection is a structure containing name and dob (which is the person's date of birth). dob contains three entries one each for date (1 ≤ date ≤ 31), month (1 ≤ month ≤ 12) and year (year > 0). Assume that each person in the person-db has a different name and none of them share date of birth. All the given dobs are valid as well.You should handle the other situations which are not mentioned here.*

*We need to find the youngest person in a given collection of persons. You should code the function get-youngest-person which takes in a person-db and returns the person who is youngest in the person-db.*
*Please code the following functions too:*

- *(get-younger person1 person2) returns the person who is younger.*

- *(empty?  person-db) returns #t if person-db is empty, #f otherwise.*

- *(first-person person-db) returns the first person in person-db.*

- *(rest-of-db person-db) returns all of person-db except the first person.*

- *(second-person person-db) returns the second person in person-db.*

```
;;here the person-db is assumed to be a list. for eg : ((vishnu (12 12 12)) (harsha (1 1 1)))
;;if the input is taken in differently it is necessary to change the selector functions in the program accordingly

(define (empty? person-db)
             (if (null? person-db)
              #t
              #f))

(define (first-person person-db)
             (car person-db))

(define (second-person person-db)
            (cadr person-db))

(define (rest-of-db person-db)
             (cdr person-db))
```

```
(define (get-dob person)
             (cadr person))

(define (get-year person)
             (caddr (get-dob person)))

(define (get-month person)
             (cadr (get-dob person)))

(define (get-date person)
            (car (get-dob person)))

(define (get-younger person1 person2)
             (cond ((> (get-year person1) (get-year person2))
 person1)
                        ((< (get-year person1) (get-year
person2))  person2)
                        ((> (get-month person1) (get-month
person2)) person1)
                        ((< (get-month person1) (get-year
person2)) person2)
                        ((> (get-date person1) (get-date
person2))  person1)
                        ((< (get-date person1) (get-date
person2))  person2)))

(define (get-youngest-person person-db)
             (cond ((empty? person-db) 'person-db-is-empty)
                        ((empty? (rest-of-db person-db))
(first-person person-db))
                        (else (get-youngest (first-person
person-db) (get-youngest-person (rest-of-db person-db))))))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: Functional Programming, Scheme, Youngest

0 Add a comment

18th September 2014

## Find Eldest

***Question:*** *Given a well formed non-empty list of items - each item being a list containing the name of an individual and his/her birth-date - we would like to get the name of the eldest. The birth-dates are three member lists where each member is an integer corresponding to: the date (1..31), month (1..12), and year (1 . . .) in that order.Sample Input : '( (ravi (12 12 12)) (ashu (11 11 191)) (mini (1 1 1911)) )*

*Output : ravi*

*You should write functions for each of the identified subtasks - like one for input date validation (do assume that there are no leap years - (29 02 2012) is therefore invalid), a function to compare two items to return the elder of the two, etc. You should not use imperative constructs like let, set, begin, list-ref etc. For every function you write, give its specification (input and output) as a one line comment above that function. You should specify the function which should be called first by naming it as main - this function should strictly follow the input-output specification given in the question.*

```
(define (date-list lst)   ;i/p-given list, o/p-list of date of
births(dob)
      (if (null? lst)
          '()
           (cons (cadar lst) (date-list (cdr lst)))))

(define (valid-date ls)   ;i/p- list of dob,  o/p- 1-valid
dobs, 0-invalid dobs
        (if (null? ls)
            1
            (if (> (caddar ls) 0)
                (if (and (>= (cadar ls) 1) (<= (cadar ls) 12))
                    (if (and (>= (caar ls) 1) (<= (caar ls)
31))
                        (if (= (date-check (car ls)) 1)
                            (valid-date (cdr ls))
                            0)
                         0)
                     0)
              0)))

(define (date-check ls)   ;i/p- dob, o/p -1 -valid, 0-invalid
         (if (and (= (remainder (cadr ls) 2) 1) (<= (car ls)
31))
             1
             (if (and (not (= (cadr ls) 2)) (= (remainder (cadr
ls) 2) 0) (<= (car ls) 30))
                 1
                  (if (and (= (cadr ls) 2) (<= (car ls) 28))
                       1
                       0))))

(define (biggest ls ls1 t t2)  ;i/p-given list,list of dobs,
first elem of dob list, first elem of given list
        (if (null? ls)
            (car t2)
            (cond ((< (caddr t) (caddar ls1)) (biggest (cdr ls)
(cdr ls1) t t2))
                  ((> (caddr t) (caddar ls1)) (biggest (cdr ls)
(cdr ls1) (car ls1) (car ls)))
                  ((< (cadr t) (cadar ls1)) (biggest (cdr ls)
(cdr ls1) t t2))
                    ((> (cadr t) (cadar ls1)) (biggest (cdr ls)
(cdr ls1) (car ls1)(car ls)))
                    ((< (car t) (caar ls1)) (biggest (cdr ls)(cdr
ls1) t t2))
                    (else (biggest (cdr ls)(cdr ls1) (car ls1)
(car ls))))))

(define (main ls) ;i/p -list, o/p-name of eldest
         (define a (date-list ls))
         (if (= (valid-date a) 0)
             'invalid-dates
             (biggest ls a (car a) (car ls))))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: Eldest, Functional Programming, Scheme

0 Add a comment

18th September 2014

## Help Shopping

***Question**:*
*Nitish Kumar (Roll No.: ?) got an S grade in the Programming Laboratory and quite happy about it, he decides to host a dinner party for his close friends. After going through his friends's choices for the different restaurants, he decides that it is better to cook the dinner. He comes with a menu and correspondingly a list of items (LOI - containing the item names and the required amounts in kg.) which need to be purchased for making those items in the menu. Being a bit of a miser and quite unaware of the price trends in Kattangal, he also prepares a price-limit list (PLL - containing the item names and the limit price) which contains the maximum price he is willing to pay for a kilogram of a particular item. Note that for some essential items, there may not be any limit price. Once he reaches the Triveni store in Kattangal, he gets the shop price list (SPP - containing the price per kg. of each of the items in LOI) and proceeds to buy the items.Given LOI, PLL and SPP, provide a bill of items as output which contains the item name, quantity bought, unit price and item's total amount for each of the items which Nitish has bought. The last entry in the list needs to be total number of items bought and total amount to be paid.*

*Example input and corresponding output:*

```
(define (step1 loi spp pll ls) ;returns a list with elements he
will be taking at the shop
         (if  (null? spp)
         ls
         (if (eqv? (list-ref (car spp) 0) (list-ref (car pll)
0))
```

```
        (if (<= (list-ref (car spp) 1) (list-ref (car pll) 1))
            (step1 (cdr loi) (cdr spp) (cdr pll) (append ls
(list (append (car loi) (cdar spp)))))
            (step1 (cdr loi) (cdr spp) (cdr pll) ls))
            (step1 (cdr loi) (cdr spp) pll (append ls (list
(append (car loi) (cdar spp))))))))

(define (step2 ls) ;returns a list including the total price of
each time
        (if (null? ls)
            '()
            (cons (append (car ls) (list (* (list-ref (car ls)
1) (list-ref (car ls) 2)))) (step2 (cdr ls)))))

(define (no-of-elem ls)
        (if (null? ls)
            0
            (+ 1 (no-of-elem (cdr ls)))))

(define (total-amount ls1)
       (if (null? ls1)
           0
           (+ (list-ref (car ls1) 3) (total-amount (cdr
ls1)))))

(define (final loi spp pll) ;main function, returns the
required list
         (define a (step2 (step1 loi spp pll '())))
         (append a (list (list (no-of-elem a) (total-amount
a)))))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: Functional Programming, Scheme, Shopping

0 Add a comment

## 18th September 2014 Xor of list elements

***Question:*** *Write a main function which computes the XOR of all elements in a list of bits. That is,*
*for example, Given L = {1 0 1 0 0 .....} find 1 XOR 0 XOR 1 XOR 0 XOR 0 .....*

```
(define (xor a1 a2)
             (cond ((= a1 a2) 0)
                       (else 1)))

(define (valid-list ls)
             (if (or (null? ls) (null? (cdr ls)))
                 0
                 1))

(define (valid-entries ls)
             (if (null? ls)
                  1
                  (if (member (car ls) '(1 0))
                       (valid-entries (cdr ls))
                       0)))

(define (xor-of-a-list ls) ;main function
             (if (= (and (valid-list ls) (valid-entries ls)) 0)
                  'list-should-have-atleast-2-elements-or-
invalid-elements
                  (if (null? (cddr ls))
                      (xor (car ls) (cadr ls))
                      (xor (car ls) (xor-of-a-list (cdr
ls))))))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: Functional Programming, Scheme, Xor

0 Add a comment

## 18th September 2014 Sum of sublist

***Question**: Given a list of sublists find the sum of each sublist*

*eg:* ( (1 2 3) (4 3 1 5) (2 0) (3 4 5) (2 3 0 0 0)) Output: (6 13 2 12 5)

```
(define sum (lambda (ls)
            (cond
            ((null? ls) 0)
            (else (+ (car ls) (sum (cdr ls)))))))
(define sum-sub (lambda (ls)
                (cond
                ((null? ls) ls)
                (else (cons (sum (car ls)) (sum-sub (cdr
ls)))))))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: foundations of programming, Functional Programming, Scheme, Sum of sublist

0 Add a comment

18th September 2014

## Above average

***Question**: Given a list of marks find the no of students above class average.*

```
(define (total ls)
        (cond ((null? ls) 0)
        (else (+ (car ls) (total (cdr ls))))))

(define (no-of-stud ls)
        (cond ((null? ls) 0)
        (else (+ 1 (no-of-stud (cdr ls))))))

(define (average ls)
        (/ (total ls) (no-of-stud ls)))

(define (no-of-stud-abv-avg ls a)
        (if (null? ls)
        0
        (if (> (car ls) a)
        (+ 1 (no-of-stud-abv-avg (cdr ls) a))
        (no-of-stud-abv-avg (cdr ls) a))))

(define (main ls)
        (no-of-stud-abv-avg ls (average ls)))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: Above average, foundations of programming, Functional Programming, Scheme

0 Add a comment

18th September 2014

## Merge sort

***Question:** Given two lists do the merge sort and the final list must be in ascending order*

```
(define (smallest ls) (small1 (car ls) ls))

(define (small1 t ls1)
        (cond ((null? ls1) t)
        ((< t (car ls1)) (small1 t (cdr ls1)))
        (else (small1 (car ls1) (cdr ls1)))))

(define (sort1 ls1)
        (if (null? ls1)
        '()
        ((let (a (smallest ls1))(cons a (sort1 (remove a
ls1)))))))

(define (remove t ls)
        (cond ((null? ls) '())
        ((eqv? t (car ls)) (remove t (cdr ls)))
        (else (cons (car ls) (remove t (cdr ls))))))

(define (combine list1 list2)
        (if (null? list1) list2
        ;else
        (if (null? list2) list1
            ;else
            (if (<= (car list1) (car list2))
                ;car of list 1 is second element of list 2
                (cons (car list1) (combine (cdr list1) list2))
                ;else
                (cons (car list2) (combine list1 (cdr
list2)))))))

(define (mergesort ls1 ls2)
        (combine (sort1 ls1) (sort1 ls2)))
```

Posted 18th September 2014 by Surya Harsha Nunnaguppala

Labels: foundations of programming, Functional Programming, Merge sort, Scheme

1 View comments

## 18th September 2014 Smallest element in a list

***Question**: Find the smallest element of a list*

```
(define (smallest a b)
        (if (a<b)
        a
        b))
(define (smallest-of-list ls)
        (cond ((null? ls) 'empty list)
        ((null? (cdr ls)) (car ls))
        (else (smallest-of-list (cons (smallest (car ls) (cadr
ls)) (cddr ls)))))
```

Posted 18th September 2014 by Vishnu Priya Matha

Labels: foundations of programming, Functional Programming, list, Scheme, Smallest element in list

0 Add a comment

## 18th September 2014 Sum of series 3

***Question**: Find the sum of the series* (1) + (1 + 2) + (1 + 2 + 3)......+ (1 + 2 + 3 + ......+ n) *where 'n' input by user*

```
(define (sum n)
        (if (= n 0)
        0
        (+ (sum2 n) (sum (- n 1)))))  ;instead of sum2
technique sum1 technique can also be used
(define (sum1 n)
        (* n (/ (+ n 1) 2)))

(define (sum2 n)
        (if (= n 0)
        0
        (+ n (sum2 (- n 1)))))
```

Posted 18th September 2014 by Vishnu Priya Matha

Labels: foundations of programming, Functional Programming, Scheme, Sum of series

0 Add a comment

## 18th September 2014 Sum of series 2

***Question**: Find the sum of the series :* x - (x^3)/3! + (x^5)/5! +......+ (x^n)/n! *where x and n are the inputs from user.*

```
(define (factorial n)
        (if (= n 0)
        1
        (* n (factorial (- n 1)))))

(define (sum1 x n count)
        (if (> count n)
        0
        (+ (/ (expt x count) (factorial count)) (sum1 x n (+
count 4)))))

(define (sum x n)
        (- (sum1 x n 1) (sum1 x n 3)))
```

Posted 18th September 2014 by Vishnu Priya Matha

Labels: Functional Programming, Scheme, Sum of series

0 Add a comment

18th September 2014

## Sum of series

***Question**: Find the sum of the series :* 1 + x + (x^2) +.....(x^n) *where x and n are the inputs*

```
(define (sum x n)
        (if (= n 0)
        1
        (+ (expt x n) (sum x (- n 1)))))
```

Posted 18th September 2014 by Vishnu Priya Matha

Labels: Functional Programming, Scheme, Sum of series

0 Add a comment

18th September 2014

## Frequency of a number in a list

***Question**: Find the frequency of a given number in a list.*

```
(define (frequency  L n)
        (cond ((null? L) count)
                  ((eqv? n (car L)) (+ 1 (frequency (cdr L) n))
                  (else (frequency  (cdr L) n ))))
```

Posted 18th September 2014 by Vishnu Priya Matha

Labels: frequency, Functional Programming, list, Scheme

0 Add a comment

18th September 2014

## Operation on datatypes

**NUMBERS**

1. (number? x) ;returns #t if x is a number, else #f
2. (complex? x) ;returns #t if x is a complex number, else #f
3. (real? x) ;returns #t if x is a real number, else #f
4. (rational? x) ;returns #t if x is a rational number, else #f

5. (integer? x) ;returns #t if x is an integer, else #f
6. (exact? x) ;returns #t if x is an exact number, else #f
7. (inexact? x) ;returns #t if x is an inexact number, else #f
8. (zero? x) ;returns #t if x is zero, else #f
9. (positive? x) ;returns #t if x is a positive number, else #f
10. (negative? x) ;returns #t if x is a negative number, else #f
11. (even? x) ;returns #t if x is a even number, else #f
12. (odd? x) ;returns #t if x is a odd number, else #f
13. (abs x) ; returns the absolute value of x
14. (acos x) ; returns the arccosine of x
15. (asin x) ; returns the arcsine of x
16. (atan x) ; returns the arctangent of x
17. (atan x1 x2) ; computes (angle (make-rectangular x2 x1))
18. (ceiling x) ; rounds x upwards
19. (cos x) ; returns the cosine of x
20. (denominator x) ; returns the denominator of x
21. (exp x) ; returns e^x
22. (expt n k) ; returns n^k
23. (floor x) ; rounds x downwards
24. (gcd x1 x2 ...) ; computes the greatest common divisor
25. (lcm x1 x2 ...) ; computes the least common multiple
26. (log x) ; returns the natural logarithm of x
27. (max x1 x2 ...) ; returns the largest of the arguments
28. (min x1 x2 ...) ; returns the smallest of the arguments
29. (modulo x1 x2) ; computes the modulo of the arguments
30. (numerator x) ; returns the numerator of the argument
31. (quotient x1 x2); returns the quotient of the arguments
32. (rationalize x1 x2); returns the simplest rational number differing from x1 no more than x2
33. (remainder x1 x2) ; returns the remainder of the arguments
34. (round x) ; rounds the argument (see the example below)
35. (sin x) ; returns the sine of x
36. (sqrt x) ; returns the square root of x
37. (tan x) ; returns the tangent of x
38. (truncate x) ; truncates x

**COMPLEX NUMBERS :**

1. (make-rectangular x1 x2) ; constructs a complex number in rectangular form
2. (make-polar x1 x2) ; constructs a complex number in polar form
3. (real-part z) ; returns the real part of the complex number z
4. (imag-part z) ; returns the imaginary part of the complex number z
5. (magnitude z) ; returns the magnitude of the complex number z
6. (angle z) ; returns the angle of the complex number z

**BOOLEAN**

1. (boolean? #t) =>#t ;returns #t if it is a boolean expression, else #f
2. (boolean? '()) => #f
3. (boolean? (eq? 'x '(x y z)) =>#t

**CHARACTERS**

1. (char? x) ;returns #t if x is a character, else #f
2. (char=? ch1 ch2) ; Is ch1 the same character as ch2?
3. (char<? ch1 ch2) ; Does ch1 come before ch2 in the alphabet?
4. (char>? ch1 ch2) ; Does ch1 come after ch2 in the alphabet?
5. (char<=? ch1 ch2) ; Does ch1 come before ch2 or are they both the same?
6. (char>=? ch1 ch2) ; Does ch1 come after ch2 or are they both the same?
7. (char-ci=? ch1 ch2) ; The same as char=? but case-insensitive
8. (char-ci<? ch1 ch2) ; The same as char<? but case-insensitive
9. (char-ci>? ch1 ch2) ; The same as char>? but case-insensitive
10. (char-ci<=? ch1 ch2) ; The same as char<=? but case-insensitive
11. (char-ci>=? ch1 ch2) ; The same as char>=? but case-insensitive
12. (char-alphabetic? ch) ;returns #t if x is an alphabet, else #f
13. (char-numeric? ch) ;returns #t if x is a numeral, else #f
14. (char-whitespace? ch) ;returns #t if x is a whitespace, else #f
15. (char-upper-case? ch) ;returns #t if x is an uppercase alphabet, else #f
16. (char-lower-case? ch) ;returns #t if x is a lowercase alphabet, else #f
17. (char-upcase ch) ;returns the uppercase of ch
18. (char-downcase ch) ;returns the lowercase of ch

**SYMBOLS**

1. (symbol? 'boo) =>#t ;returns #t if given argument is a symbol, else #f
2. (symbol? "boo") =>#f
3. (symbol? (car '(a b c))) =>#t
4. (symbol? '()) =>#f
5. (symbol->string 'foo) =>"foo" ;converts given symbol into a string and returns the string
6. (string->symbol "foo") =>foo ;converts given string into a symbol and returns the symbol
7. (symbol->string (string->symbol "bar"))=>"bar"

### Operations on strings

- (string? str) ;returns true if it is a string else false
- (string=? str1 str2) ; Is str1 the same as str2?
- (string<? str1 str2) ; Is str1 lexicographically less than str2?
- (string>? str1 str2) ; Is str2 lexicographically less than str1?
- (string<=? str1 str2) ; Is str1 lexicographically less than or equal to str2?
- (string>=? str1 str2) ; Is str2 lexicographically less than or equal to str1?
- (string-ci=? str1 str2) ; The same as string=? but case-insensitive
- (string-ci<? str1 str2) ; The same as string<? but case-insensitive
- (string-ci>? str1 str2) ; The same as string>? but case-insensitive
- (string-ci<=? str1 str2) ; The same as string<=? but case-insensitive
- (string-ci>=? str1 str2) ; The same as string>=? but case-insensitive
- (make-string n) ;to create strings
- (make-string n ch) ;to create strings

examples:

```
> (make-string 10 #\a)

"aaaaaaaaaa"

> (string #\S #\c #\h #\e #\m #\e)

"Scheme"
```

- (string-length str) ;returns the legth of the string str
- (string-append str1 str2 str3....) ;to concatenate strings
- (string-ref str1 pos) ;returns the element at the position pos in str1
- (substring str start end) ;str must be a string and start and end are integers such that the length of the string
- (string->list str) ;convert strings to list
- (list->string list1) ;converts list to string
- (string-copy str) ; to copy strings, i.e gives output as atr
- (string-set! str n ch) ;stores ch in element n of str. The value returned is unspecified.
- (string-fill! str ch) ; stores ch in every element of str as a side effect. The return value is unspecified.

### Operations on pairs and lists

- (length list1) ;returns length of list1
- (pair? x) ;returns true if it is a pair else false
- (list? x)

- (null? x)
- (list? list1) ;returns #t if list1 is a list else #f
- (null? list1) ;null predicate
- (list-ref listref pos);to refer an element in list in the position pos
- (reverse list1) ;returns the reverse of the list1
- (append list1 list2) ;returns a list with list2 elements added st the end of list1
- (memq x ls)     ;uses eq?
- (memv x ls)     ;uses eqv?
- (member x ls)   ;uses equal?

### MAP

(map procedure list1) ;applies procedure to all elements of list

**Note that map only works on the top-level items of a list.**

If two or more lists are given as arguments to map, the procedure will be applied to each *n*th element of the lists

Posted 18th September 2014 by Vishnu Priya Matha

Labels: basics, foundations of programming, Functional Programming, operations in datatypes, Scheme, syntax

0 Add a comment

18th September 2014

## Some basics of Scheme

**IDENTIFIERS**

Identifiers can be formed by using

1. a to z
2. A to Z
3. 0 to 9
4. ? - ! + . * / < > = : $ % & _ ~ @

**COMMON NOTATIONS**

; Comment i s written after a semicolon and it is till the end of the line.
true- #t
false- #f

-

**GETTING STARTED**

Scheme follows read-evaluate-print loop.

```
"hi mom"
>hi mom
42
>42
(+ 13 12)
>25

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

(car '(d g r))
>d

(cdr '(f r g))
>(r g)

(cons 'a '(b c))
>(a b c)

(cons (car '(a b c)) (cdr '(d e f)))
>(a e f)
```

## define

Syntax:

```
 (define <variable> <expression>)
```

eg: defining procedure

```
 (define square
     (lambda (n) (* n n)))
 (square 5)
>25
```

## lists

In scheme lists needn't contain the same type of object. In the following manner any number of arguments can be given to a function:

```
 (define (sum (x.y))(+ x y))
(sum 1 2 3)
>6
(sum 1 2 3 4)
>10
```

Example of list of strings:

```
(define a (list "this" "is" "a" "list"))
a
>(this is a list)
```

## QUOTE AND QUASIQUOTE

```
(quote (+  4 5))
>(4 5)
```

Quote forces the given list to be treated as data.It is also used to avoid scheme from treating words as variables.

Syntax of quasiquote:
(quasiquote <qq template>)

```
 '(6 ,(+ 2 3) 10)
>(6 5 10)
```

The "," here unquotes the data in the paranthesis and evaluates it and the result is given back which forms an element of the list.

**CAR AND CDR**

car and cdr functions are inbuilt functions in scheme.car of a list returns the first element of a list while cdr of a list returns the list excluding the first element. (refer to examples given in section 4).

**CONS**

Cons is used to construct a list and its example is given in section 4.

**LET EXPRESSIONS**

**let**

Syntax:

```
(let  ((var1 exp1)
       (var2 exp2)
        . . .
       (varn expn))
 body)
```

Examples:

```
(let  ((x 12) (y 10))
      (+ x y))
>22
```

Also, variables bound by let are let bound variables and visible only in let.

```
(define x 10)
(let ((x 5) (y (* x 2)))
      (+ x y))
>25
```

The variable *x* will be bound to the value 5 and the variable *y* will be bound to 20, not to 10. This is because the *x* in the expression for *y* is computed outside the let expression and the value of that *x* is 10.

**let***

The syntax of let* is similar to that of let but semantics is as follows:

```
(let ((var1 expr1))
      (let ((var2 expr2))
        (let ...
          (let ((varn exprn))
   body) ... )))
```

Example:

```
(define x 10)
(let*  ((x 5)
       (y (* x 2)))
    (+ x y))
> 15
```

**named let**

It has the following syntax:

```
  (let var bindings body)
```

Example:

```
(define factorial
  (lambda (n)
    (let iter ((product 1)
              (counter 1))
      (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))))
```

So by using named let it is possible to invoke the body again and again and do the recursion.

**letrec**

Syntax:

```
(letrec ((var1 val1)
        (var2 val2)
         . . .
        (varn valn))
 body)
```

Example:

```
(define factorial
 (lambda (n)
   (letrec ((iter
             (lambda (product counter)
               (if (> counter n)
                  product
                 (iter (* counter product) (+ counter 1))))))
 (iter 1 1)))
```

**CONDITIONAL EXPRESSIONS**

### COND

Syntax:

```
(cond (condition1 consequent1)
         (condition2 consequent2)
          . . .
        (else alternative))
```

### CASE

Syntax:

```
(case key clause1 clause2 ...)
```

The key may be an expression and clause should have the following syntax:

```
 ((datum1 ...) expression1 expression2 ...)
```

### PREDICATES

Predicates are those which return a true or false value.

Example:

```
(number? 12)
>#t
```

Scheme has a lot of predicates predefined and some of them are null?, string?, pair?, char?, etc..

### PROCEDURE TO REMOVE AN ELEMENT FROM LIST

```
(define remove
  (lambda (x ls)
    (cond ((null? ls) '())
             ((eqv? (car ls ) x) (remove x (cdr ls)))
             (else (cons (car ls) (remove x (cdr ls)))))))

(remove 'b '(b d c d b))
>(d c d)
```

### PROCEDURE TO FIND THE TAIL OF THE LIST FROM GIVEN OBJECT

```
(define memv
     (lambda (x ls)
         (cond ((null? ls) #f)
                  ((eqv? (car ls) x) ls)
                  (else (memv x (cdr ls))))))

(memv 'b '(a b c d b e f))
>(b c d b e f)
```

Posted 18th September 2014 by Vishnu Priya Matha

Labels: basics, Functional Programming, reference, Scheme

0 Add a comment

## 16th September 2014

## Introduction

Scheme is a semi-functional programming language and a dialect of Lisp, which stands for LISt Processing. It is regarded as one of the most elegant programming languages in the world thanks to its smallness, beauty and expressive power.Scheme is very expressive and several programming paradigms. Syntactically scheme is very simple and beautiful. This site gives the basics of scheme and gives the code of most common programs written.Scheme programs can be run on linux terminal by installing mit/gnu scheme from the site http://www.gnu.org/software/mit-scheme/ [http://www.gnu.org/software/mit-scheme/] .

Posted 16th September 2014 by Vishnu Priya Matha

Labels: Functional Programming, Scheme

0 Add a comment