Programming Language Homework 1 Report

張財實
資訊三乙
F74055047

Problem1

P1.1

```
(defun maxdivisor(n)
  (multiple-value-bind (value)(floor (sqrt n))
value))

(defun prime(x)
(cond
  ((= x 1) (print Nil))
  ((= x 2) (print T))
  (T (let ((n (maxdivisor x)))
    (loop for i from 2 to n
      when (= (mod x i) 0)
      do (return (print Nil))
      when (and (not (= (mod x i) 0)) (= i n))
      do (return (print T)))))))
```

My approach:
0
I am using square root approach to solve prime number.

The reason it is:
n = a*b and a <= b then a*a <= a*b = n

So we do not have to check all the divisor from 2 to n , we just need to check whether it can be divisible by any value from 2 to sqrt(n)

One way is to check divisible is to use MOD macro.

P1.2

```
(defun palindrome(input)
  (if (equal (reverse input) input) (print T)
(print Nil)))
```

Palindrome is relatively easy than P1.1

As the tip said, just reverse the list and check equality will do.

P1.3-1

```
(defun fib1(n)
  (cond
    ((= n 0) 0)
    ((= n 1) 1)
    (T (+ (fib1 (- n 1)) (fib1 (- n 2))))))

(format t "~%")
(trace fib1)
(fib1 3)
```

Just like every fib in c, I make 0^{th} and 1^{st} fib number special cases and the rest is normal recursive call.

P1.3-2

```
(defun tailfib(n a result)
  (if (= n 1) result (tailfib (- n 1) result (+
result a))))

(defun fib2(n)
(if (= n 0) 0 (tailfib n 0 1)))

(trace fib2)
(fib2 8)
```

In my opinion, the tail recursion is just a recursion call does what a normal for loop does except that for loop doesn't require to call itself again and again.

So, I added two more parameter 'a' and 'result', renew the 'a' by previous 'result', 'result' renew itself by adding current 'a' to itself.

Output for problem1 script:

```
PROBLEM1.1-TEST-CASES
T
T
NIL
T
PROBLEM1.2-TEST-CASES
NIL
T
NIL
T
T
PROBLEM1.3-TEST-CASES
TRACE-CASE-FOR-NORMALFIB
 0: (FIB1 3)
  1: (FIB1 2)
   2: (FIB1 1)
   2: FIB1 returned 1
   2: (FIB1 0)
   2: FIB1 returned 0
  1: FIB1 returned 1
  1: (FIB1 1)
  1: FIB1 returned 1
 0: FIB1 returned 2

TRACE-CASE-FOR-TAILFIB
 0: (FIB2 8)
 0: FIB2 returned 21
```

Problem2

```
(defun split-number (numbers)
  (ceiling (/ (length numbers) 2)))

(defun left-split (numbers l)
  (cond ((= l 0) '())
    (T (cons (car numbers) (left-split (cdr
numbers) (- l 1))))))

(defun right-split (numbers l)
    (cond ((= l 0) numbers)
      (T (right-split (cdr numbers) (- l 1)))))

(defun merge-two-list (left right)
  (cond ((not right) left)
    ((not left) right)
    ((< (car left) (car right)) (cons (car left)
(merge-two-list (cdr left) right)))
    (T (cons (car right) (merge-two-list left
(cdr right))))))

(defun mergesort (numbers)
  (if (= (length numbers) 1) numbers
    (merge-two-list
      (mergesort (left-split numbers (split-
number numbers)))
```

split-number: Tell mergesort which is the boundary of split

left/right-split: Recursively split the number to desired list

merge-two-list: Merge two list to one, sorting them recursively

mergesort: Main call of mergesort, return single-value-list to merge-two-list or split the numbers

```
      (mergesort (right-split numbers (split-
number numbers))))))

; main function
(let
  ((n (read))(numbers))
  (setf numbers
    (do ((i 0 (+ i 1))
      (tmp nil))
      ((>= i n)
      (reverse tmp))
      (setf tmp (cons (read) tmp))))
      (format t "~{~A ~}~%" (mergesort
numbers)))
```

Output for problem2 script: (Traced)

$ sbcl --script problem2.lsp
3
3 2 1

1 2 3

$ sbcl --script problem2.lsp
5
1 3 8 9 1

1 1 3 8 9

$ sbcl --script problem2.lsp
10
9 8 16 2 7 199 0 98 1 29

0 1 2 7 8 9 16 29 98 199

Problem3

```
(defun file-to-list (file1 file2)
  (values
    (with-open-file (stream file1)
       (loop for line = (read-line stream nil)
          while line
          collect line))
    (with-open-file (stream file2)
      (loop for line = (read-line stream nil)
        while line
        collect line))))

(defun result (list1 list2 l r)
  (cond
```

file-to-list: Using with-open-file to open the file securely and read every lines into a list. A quick way to return two list is by VALUES.

result: To recursively stack the answer to the answer stack(desired output list). At this stage, we have already known how many lines from the left and right lists have to be cut into the result list. If there is a match

```
    ((> l 0) (cons (format nil "~c[31m- ~a~%~c[0m" #\ESC
(car list1) #\ESC) (result (cdr list1) list2 (- l 1) r)))
    ((> r 0) (cons (format nil "~c[32m+ ~a~%~c[0m" #\ESC
(car list2) #\ESC) (result list1 (cdr list2) l (- r 1))))
    ((and (= l 0) (= r 0) (car list1) (car list2)) (cons
(format nil " ~a~%" (car list2)) (merge-list (cdr list1)
(cdr list2) (cdr list2) 0)))))

(defun merge-list (list1 list2 list2-copy r)
  (cond ((and (not list1) (not list2-copy)) '())
    ((not list2-copy) (let ((l (length list1)))
      (result list1 list2 l r)))
    ((not (position (car list2-copy) list1 :test #'equal))
(merge-list list1 list2 (cdr list2-copy) (+ r 1)))
    ((position (car list2-copy) list1 :test #'equal)
      (let ((l (position (car list2-copy) list1 :test
#'equal))) (result list1 list2 l r)))))

(defun diff (file1 file2)
  (multiple-value-bind (list1 list2)
    (file-to-list file1 file2)
    (format t "~{~A~}" (merge-list list1 list2 list2 0))))

(format t "~%Output:~%")
(diff "file1.txt" "file2.txt")
```

it should return to '**merge-list**' function.

merge-list: At this stage, we assume the diff command is right file align. Which means that before RHS line meets his match, the LHS will be all unmatched (----…..) and RHS should be iterate thru up and down, if no matches, it should be (++++…..), so **merge-list** is to record how many '+' and '-' to call the '**result**' function to make the list.

diff: Required function.

Problem 3 algorithm detail:

eg:
list1: a 1 b 3 5 maps to the file1.txt
list2: c d 1 e 3 5 maps to the file2.txt

L	R	list1	list2	list2-copy	stack
0	0			merge-list	-
1	2	a 1 b 3 5	c d 1 e 3 5	c d 1 e 3 5	-
0	0	result	result	merge-list	-
1	1	b 3 5	e 3 5	e 3 5	-a+c+d1
0	0	result	result	merge-list	-a+c+d1
0	0	5	5	5	-a+c+d1-b+e
0	0	result	result	merge-list	-a+c+d1-b+e
0	0	nil	nil	nil	-a+c+d1-b+e5

Output for problem3 script:

$ sbcl --script problem3.lsp

```
- #include <stdio.h>
+ #include <iostream>
+ using namespace std;
 int main() {
-     printf("Hello World");
+     cout << "Hello World" << endl;
     return 0;
 }
```