Practice Problem Set 4

CS 152 – Abstractions and Paradigms in Programming

Date - 1st March 2013

1. Write a function to-decimal to convert a list of characters into decimal numbers. Your number should, for instance, convert ‘(#\2 #\4 #\3 #\. #\6 #\5) into 243.65.

   Write your program in both the functional and the imperative style (using set!). If you think some variable represents the state of the system, name the variable as state.

   Can you write the imperative program using a foldr or foldl?

2. Consider a non-terminating system which keeps on accepting a sequence of numbers as inputs from the keyboard using the function called (read). read about read (pun intended) from the racket documentation. The system, on receiving an input, (display)s the sum of the last three inputs read.

   Write a function called which simulates the system. Once again, clearly identify the state variable.

3. Consider a gate leading out of a parking lot. The gate has three sensors:

   -gatePosition has one of three values ‘top, ‘middle, ‘bottom, signifying the position of the arm of the parking gate.

   -carAtGate is True if a car is waiting to come through the gate and False otherwise.

   -carJustExited is True if a car has just passed through the gate; it is true for only one step before resetting to False.

   The gate has three possible outputs (think of them as controls to the motor for the gate arm): ‘raise, ‘lower, and ‘nop. (Nop means no operation.)

Roughly, here is what the gate needs to do:

- If a car wants to come through, the gate needs to raise the arm until it is at the top position.

- Once the gate is at the top position, it has to stay there until the car has driven through the gate.

- After the car has driven through the gate needs to lower the arm until it reaches the bottom

Write a function called parking-lot-system which simulates the system. Once again, clearly identify the state variable.

4. Consider the bank account objects created by make-account, with the password modification. Suppose that our banking system requires the ability to make joint accounts. Define a procedure make-joint that accomplishes this. make-joint should take three arguments. The first is a password-protected account. The second argument must match the password with which the account was defined in order for the make-joint op eration to proceed. The third argument is a new password. make-joint is to create an additional access to the original account using the new password.

For example, if peter-acc is a bank account with password open-sesame, then

```
(define paul-acc (make-joint peter-acc 'open-sesame 'rosebud))
```

will allow one to make transactions on peter-acc using the name paul-acc and the password rosebud. The original make-account is given below:

```
(define (make-account balance)
 (define (withdraw amount)
   (if (>= balance amount)
       (begin (set! balance (- balance amount))
              balance)
       "Insufficient funds"))
 (define (deposit amount)
   (set! balance (+ balance amount))
   balance)
 (define (dispatch m)
```

```
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else "Unknown request")))
  dispatch)
```

5. A tool frequently used while developing software is a profiler. A simple form of profiler counts the number of times a given function is called during the course of a computation. If we have this information, then we can try very hard to ensure that frequently called functions are written efficiently.

Write a function (make-monitored f) that takes as input a function f and returns a function, say mf, that keeps track of the number of times it has been called, by maintaining a counter. If the input to mf is the special symbol how-many-calls?, then mf returns the value of the counter. If the input is the special symbol reset-count, then mf resets the counter to zero. For any other input, mf returns the result of calling f on that input and increments the counter. For instance, we could make a monitored version of the sqrt function:

```
> (define s (make-monitored sqrt))
> (s 100)
10
> (s 25)
5
> (s 'how-many-calls?)
3
> (define p (make-monitored +))
> (p 23 45)
68
> (p 23 46)
69
> (p 23 47)
70
> (p 'how-many-calls?)
3
> (p 'reset-counter)
done
> (p 'how-many-calls?)
0
```

6. If the language is without side-effects, we do not have to specify the order in which the subexpressions should be evaluated (e.g., left to right or right to left). When we introduce assignment, the order in which the arguments to a procedure are evaluated can make a

difference to the result. Define a simple procedure f such that evaluating (+ (f 0) (f 1)) will return 0 if the arguments to + are evaluated from left to right but will return 1 if the arguments are evaluated from right to left.

```
> (+ (f 0) (f 1))
0
```

Since DrScheme evaluates expressions from left to right, I shall simulate a right-to-left evaluation by flipping the arguments:

```
> (+ (f 1) (f 0))
1
```

A second time evaluation also produces the same result

```
> (+ (f 1) (f 0))
1
> (+ (f 0) (f 1))
0
>
```

7. Using the environmental model of execution, explain the
output of the following program: \hspace*{1cm} \hfill (8 Marks)

```
(define (f i q)
  (define (p) @i)
  (if (> i 3) (f (-i 2) p)
      (begin
        (+ (p)(q)))))
(define (g) ())
(define (m) (f 4 g))
(m)
```

Your explanation should include the figures of the environment at the point marked with @ every time it is visited.

8. Memoize the following function. Called the memoised function memo-choose.

```
(define (choose n r)
  (if (or (= r 0) (= r n))
      1
      (+ (choose (- n 1) (- r 1))
         (choose (- n 1) r))))
```

Consider the calls (choose 5 3) and (memo-choose 5 3). How many calls are avoided as the result of memoization?
\item Here is a solution of the minchange problem reproduced:

9.

```
(define infty 99999999999999)
(define (minchange m)
    (if (= m 0)  0
    (min (if  (>= m  50) (+ 1 (minchange (- m 50))) infty)
         (if  (>= m  25) (+ 1 (minchange (- m 25))) infty)
         (if  (>= m  20) (+ 1 (minchange (- m 20))) infty)
         (if  (>= m  10) (+ 1 (minchange (- m 10))) infty)
         (if  (>= m  5) (+ 1 (minchange (- m 5))) infty)
         (if  (>= m  3) (+ 1 (minchange (- m 3))) infty)
         (if  (>= m  2) (+ 1 (minchange (- m 2))) infty)
         (+ 1 (minchange (- m 1))))))
```

Produce a memoised version of the function. That is, write a function memoise-minchange which will take a parameter n and produce a memoised version of minchange for the argument range 0 to n.

Let us define memo-minchange as (memoised-minchange 500). Now suppose I called memo-minchange as (memo-minchange 10). How many calls to memo-minchange would be made?