Tutorial Problem Set 4

CS 152 – Abstractions and Paradigms in Programming

1. 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.

2. 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`. If you think some variable represents the state of the system, name the variable as `state`.

3. Modify the `make-account` procedure so that it creates password-protected accounts. That is, `make-account` should take a symbol as an additional argument, as in `(define acc (make-account 100 'secret-password))`. The resulting account object should process a request only if it is accompanied by the password with which the account was created, and should otherwise return a complaint:

```
((acc 'secret-password 'withdraw)  40)
>  60
((acc  'some-other-password 'deposit)  50)
"Incorrect password"
```

4. Consider the bank account objects created by `make-account`, with the password modification described in the previous question. 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` operation 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`. You may wish to modify your solution to the previous question to accommodate this new feature.

5. If the language is without side-effects, we do not have to specify the order in which the sub-expressions 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 drracket 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
>
```

6. **Pen and paper exercise:** Using the environmental model of execution, explain the output of the following program:

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

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

7. Consider the following program:

```
(define (f x)
  (define (p f) (f (* x x)))
  (define (g y)
    (cond [(= x 2) (begin (set! x (+ x 2))
                          (h (- y 1)))]))
  (define (h x)
    (p (lambda (w) (+ w x))))
  (g x))
(f 2)
```

In the environmental model of execution, show the *complete environment* just when the expression `(+ w x)` is about to be evaluated. By the complete environment, we mean all the frames that are created and their contents, and, for each frame, its global pointer. Also mention against each frame, the name of the corresponding function (or the lambda, if it is an anonymous function).

8. Memoize the following function.

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

Call the memoised function `memo-choose`.