Tutorial Problem Set 4

CS 152 – Abstractions and Paradigms in Programming

Announcement Date: 17th March 2017 Submission Date: 24th March 2017

1. Write a macro to implement C-style while loops in drracket. In general, the syntax of while is:

   ```
   (while boolean-expression one-or-more-statements)
   ```

   The example below illustrates the use of the while macro:

   ```
   (define i 10)
   (define y 0)
   (while (> i 0)
      (set! y (+ y i))
      (set! i (- i 1)))
   ```

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. 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`.

4. 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 `(sum-of-last-three)` which simulates the system. Once again, clearly identify the state variable.

5. 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 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`. 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)
```

6. 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
>
```

7. **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.