Homework: DefineLang and FuncLang

Learning Objectives:

1. Write programs in DefineLang, FuncLang
2. Get familiar with the concepts of recursive functions, high order functions, currying

Instructions:

- Total points: 39 pt.
- Early deadline: Oct 2 (Wed) 2018 at 11:59 PM; Regular deadline: Oct 4 (Fri) 2018 at 11:59 PM (you can continue working on the homework till TA starts to grade the homework).
- Write your Definelang and Funclang programs for the following questions and submit them in one pdf file. You can use Racket to test your written funclang programs as you start your homework.
- You can reuse any functions you write in this homework to answer questions.
- Funclang interpreter will be released on Sep 30 (Mon), then, you can use your Funclang interpreter provided in hw4code.zip to test the correctness of the programs. Follow the steps in the tutorial of the homework 2 to setup the interpreter.
- How to submit:
  - Submit your programs to Canvas under Assignments, HW4 Early/Regular Submission.
  - Please provide the complete solutions in one pdf file.

Questions:

1. (3 pt) Write a DefineLang program to define the ASCII values for the letters 'X' 'Y' and 'Z', and then compute the ASCII values for their corresponding lower case letters.

   Sol:

```
(define X 88)
(define Y 89)
(define Z 90)
(define x (+ X 32))
(define y (+ Y 32))
(define z (+ Z 32))
```

2. (3 pt) Write a function f to compute the n^{th} number in Fibonacci Sequence.

```
$ (f 1)
1
$ (f 2)
1
$ (f 3)
2
```

Sol:

```
(define f
        (lambda (n)
                (if (= n 1)
                        1
                        (if (= n 2)
                                1
                                (+ (f (- n 1)) (f (- n 2)) )
                        )
                )
        )
)
```

3. (12 pt) FuncLang programming: list.

   (a) (4 pt) Write a function *count* that counts the occurrence(s) of a given element n in a list L

```
$ (count 1 (list))
0
$ (count 10 (list 1 10 3 14))
1
$ (count 11 (list 11 11 31 14))
2
```

   Sol:

```
(define count
        (lambda (n lst)
                (if (null? lst)
                        0
                        (if (= (car lst) n)
                                ( + 1 (count n (cdr lst)) )
                                (count n (cdr lst))
                        )
                )
        )
)
```

   (b) (4 pt) Write a function *max* that identifies the maximum value of a list

```
$ (max (list))
0
$ (max (list 1 10 3 14))
```

14
$ (max (list 11 18 31 14))
31

Sol:

```
(define max
        (lambda (lst)
                (if (null? lst)
                        0
                        (maxhelper lst (car lst) )
                )
        )
)

(define maxhelper
        (lambda (lst max)
                (if (null? lst)
                        max
                        (if (> (car lst) max)
                                (maxhelper (cdr lst) (car lst))
                                (maxhelper (cdr lst) max )
                        )
                )
        )
)
```

(c) (4 pt) Write a function *maxrepeat* that returns the frequency of the most frequently occurred element in a list
$ (maxrepeat (list))
0
$ (maxrepeat (list 1 10 3 14))
1
$ (maxrepeat (list 11 11 31 14))
2

Sol:

```
(define maxrepeat
        (lambda (lst)
                (if (null? lst)
                        0
                        (maxrepeathelper lst 1)
                )
        )
)
(define maxrepeathelper
        (lambda (lst max)
                (if (null? lst)
                        max
                        (if (> (count (car lst) lst) max)
                                (maxrepeathelper (cdr lst) (count (car lst) lst))
                                (maxrepeathelper (cdr lst) max)
                        )
                )
        )
```

```
)
```

4. (5 pt) FuncLang programming: list and pair.

   (a) (2 pt) Using list expression define a list named *pairs* that contains a list of 4 pairs: (1,5) (6,4) (7,8) (15,10).

   Sol:

```
(define pairs (list (cons 1 5) (cons 6 4) (cons 7 8) (cons 15 10)))
```

   (b) (3 pt) Define a function secondSum that performs an addition on the second element of each pair on the list *pairs*. The function has to extract each element using list-related expressions. Result = 5 + 4 + 8 + 10 = 27.

   Sol:

```
(define secondSum
        (lambda (lst)
                (+ (cdr (car pairs)) (cdr (car (cdr pairs))) (cdr (car (cdr (cdr pairs)))) (cdr (car (cdr (cdr (cdr
                    pairs))))))
        )
)
```

5. (8 pt) Given the following definitions of *pair* and *apair*

   (define pair (lambda (fst snd) (lambda (op) (if op fst snd))))
   (define apair (pair 2 3))

   (a) (2 pt) Write a function *second* to return the second element of *apair*
   Sol:

```
(define second
        (lambda (p) (p #f)
        )
)
```

   (b) (2 pt) Modify *pair* to support arithmetic between two elements of *apair*

   Sol:

```
(define pair
        (lambda (fst snd)
                (lambda (op)
                        (if (= op 0) (+ fst snd)
                                (if (= op 1) (- fst snd)
                                        (if (= op 2) (* fst snd)
```

```
                                        (if (= op 3) (/ fst snd) 0
                                        )
                                )
                        )
                )
        )
)
```

(c) (4 pt) Write a FuncLang program to return the addition of the two elements of *apair*

Sol:

```
(define add
        (lambda (p) (p 0)
        )
)
```

6. (8 pt) FuncLang programming: high order functions and curried functions.

(a) (2 pt) Construct a global variable mylist that holds a list of three pairs, (1,3) (4,2) (5,6).

Sol:

```
(define mylist (list (cons 1 3) (cons 4 2) (cons 5 6)))
```

(b) (4 pt) Write a function applyonnth that takes three arguments op, lst, n, where op is a function, lst is a list of pairs, n is an integer. The return value should be the result of applying op on the n-th pair in the list. If n is out of range of the list, return -1. You can assume op is a function valid to accept two arguments.

Some examples of using applyonnth with above mylist variable:
$ (applyonnth add mylist 1)
4 // 1+3
$ (applyonnth subtract mylist 2)
2 // 4-2
$ (applyonnth add mylist 8)
- 1 // third parameter out of range
$ (applyonnth add mylist -1)
- 1 // third parameter out of range

Sol:

```
(define applyonnth
        (lambda (op lst n)
                (if (null? lst)
                        -1
                        (if (= n 1)
                                (op (car (car lst)) (cdr (car lst)))
                                (if (< n 0)
                                        -1
```

```
                                        (applyonnth op (cdr lst) (- n 1))
                                )
                        )
                )
        )
)
```

(c) (2 pt) Convert the above FuncLang program into the curried form.

Sol:

```
(define applyonnth
        (lambda (op)
                (lambda (lst)
                        (lambda (n)
                                (if (null? lst)
                                        -1
                                        (if (= n 1)
                                                (op (car (car lst)) (cdr (car lst)))
                                                (if (< n 0)
                                                        -1
                                                        (((applyonnth op) (cdr lst)) (- n 1)))
                                                )
                                )
                        )
                )
        )
)
```