Lab 6

77 Points Total – including 10 points for attendance

LAB/HOMEWORK GOALS

This week's lab will help you practice:

- Working with Lists
- Working with self-referential data definitions
- Lists of Structures
- Lists of enumerated types
- Lists of Lists
- List abbreviations

You will need to change your Dr. Racket language level. In the lower left, click the selection for "Beginning Student" and change it to "Beginning Student with List Abbreviations".

Question 1 – (52 Points)

You have been hired by an online shopping company to check password security of its customers. In an effort to keep passwords secure, you have decided that a valid password must

(1) be at least 6 characters long and not more than 10 characters long

(2) contain at least one capital letter

(3) contain at least one small letter

(4) contain at least one of the following special characters: #,$,%,!,&

You need to write some code that that will check passwords and generate strings indicating what violations have occurred (if any). The following data definitions will be used:

```
; a Password-error-string is one of
;-- "Bad Length!"
;-- "No capital letter!"
;-- "No small letter!"
;-- "No special character!"

; a List of Password-error-strings (LoPES) is
; -- empty
; -- (cons Password-error-string LoPES)

(define-struct checked-password (pswd error-list))
; a checked-password is a (make-checked-password string LoPES)
; INTERP
```

; pswd is the password itself
; error-list is a list of strings - one for each error the password
; exhibits

(define CPWD1 (make-checked-password "Agood1!" empty))
(define CPWD2 (make-checked-password "sm#" (list "No capital letter!"
"Bad Length!")))

1.1 (10 points) Some Preliminaries: string->LO1S and LO1S->string

In order to work with passwords, you will need to take a string and break it into a list of 1Strings (recall that a 1String is a string of length 1). While there is a built-in function in racket that will do this job (explode) and to convert back again (implode) – here you are asked to write your own versions. The following data definitions will be useful.

; a 1String is a string of length 1

; a List of 1String (LO1S) is
; - empty
; - (cons 1String empty)

We can have a self-referential definition of a string as well:

; a String is
; - “” ; the empty string
; - (string-append 1String String)

1.1a. Design and implement the function string->LO1S that consumes a string and produces a list of 1Strings. This function should work like the explode function that is in Dr. Racket. For example (you will need to provide additional check-expects to write the function):

(check-expect (string->LO1S "this!") (list "t" "h" "i" "s" "!"))

1.1b. Design and implement the function LO1S->string that consumes a list of 1Strings and produces a string made out of those 1Strings. This function should work like the implode function that is in Dr. Racket. For example (you will need to provide additional check-expects to write the function):

(check-expect (LO1S->string (list "W" "O" "W" "I" "E" "!" "!")) "WOWIE!!")

1.2 (20 points) Design and implement 4 functions that will check the 4 rules (see below for how these functions are called). These functions will each either return the boolean true (if the password conforms to the rule) or it will return an error string. These functions will take either a string representing the password, or a list of 1strings that is the result of calling the explode function on that string. You may find the built-in functions string-lower-case? and string-upper-case? helpful. The signatures for the functions are the following:

; check-length: string -> boolean or Password-error-string
; check-for-1-cap: list-of-1string-> boolean or Password-error-string
; check-for-1-sm: list-of-1string -> boolean or Password-error-string

; check-for-sp-char: list-of-1string -> boolean or Password-error-string

These functions will be used in the following function that has been implemented for you which is shown here:

```
; string->checked-password: string -> checked-password
; write a function that given a string representing a password
; returns the corresponding checked-password structure
(define (string->checked-password apword)
  (make-checked-password
   apword
   (filter-strings (list
                    (check-length apword)
                    (check-for-1-cap (explode apword))
                    (check-for-1-sm (explode apword))
                    (check-for-sp-char (explode apword))))))
```

1.3 (5 points) Notice there is a function filter-strings referenced. You must write that function. Notice that the list constructed in the function will contain true in cases where the test passed, and an error string for cases that the test did not pass. The following data definitions and signature should be used in your implementation.


```
; a boolean-or-string list (BoSL) is
; -- empty
; -- (cons boolean BoSL)
; -- (cons string BoSL)

; filter-strings: BoSL -> LOS
; consumes a list containing both booleans and strings
; and produces the list of all strings in the original list


(check-expect (string->checked-password "Agood1!") CPWD1)
(check-expect (string->checked-password "sm#") CPWD2)
```

1.4 (8 points) Write the function check-list-of-passwords that consumes a list of strings representing passwords and produces the corresponding list of checked-password structures.

```
(check-expect (check-list-of-passwords (list "Agood1!" "sm#"))
              (list CPWD1 CPWD2))
```

1.5 (9 points) Write the function list-failed-check-passwords that takes a list of strings representing passwords and produces a list of checked-passwords structures containing just those structures for passwords that fail at least one of the password rules.

```
(check-expect (list-failed-check-passwords (list "Agood1!" "sm#"))
              (list CPWD2))
```

Question 2 – More fun with Lists: flatten (15 Points)

Consider the following definition of a list whose elements may either be numbers or lists (whose elements may be either numbers or lists whose…).

; a list of numbers or lists (LoNoL) is
; - empty
; - (cons number LoNoL)
; - (cons LoNoL LoNol)

For example,

(define L0 empty)
(define L1 (list 1 2 3 4 5))
(define L2 (list (list 1 (list 2) 3) (list (list 4 5))))

Design and implement the function flatten that takes a LoNoL and produces a list of numbers containing the numbers in the original list. For example (you will need to write more check-expects to write your function):

(check-expect (flatten L0) empty)
(check-expect (flatten L1) L1)
(check-expect (flatten L2) L1)

Helpful Function: You will find it useful to use the **append** function that is in racket. Append consumes 2 arguments which should be lists, and produces a new list containing all of the elements of the first followed by the elements of the second. For example the following two check-expects highlight the difference between cons and append:

(check-expect (**cons** (list 1 2) (list 3 4)) (list (list 1 2) 3 4))
(check-expect (**append** (list 1 2) (list 3 4)) (list 1 2 3 4))