Homework 2: Getting to Know Racket

Due September 20th at 10pm

There are two steps to submitting this assignment:

(1) Place your write-up in a Google Drive folder that is shared with me. Call it FIRSTNAME LASTNAME Homework 2.

(2) Fill out the Homework 2 Submission Form.

Part 1: PL concepts (30 points)

1. side effects (10 points)

A. In class, we described a side effect as "a behavior other than producing a value". Do you think variable declaration in normal languages is a side effect? Why or why not?

B. Now consider the wacky language shown in the following code snippet:

```
def foo():
    x = 5
    return
foo()
print(x)                # prints 5
```

As you can see, this language has weird scope: variables declared inside a function are available after the function has returned.

Does this example change your answer above? Why or why not?

2. if (20 points)

Conditionals are one of the most basic programming language constructs. Conventionally, they take three arguments: the *predicate*, the *true branch*, and the *false branch*. But different programming languages make design choices about conditionals.

Explore how **if**-conditionals are implemented in Racket. Answer the following questions:

(1) What kinds of values can be supplied as the predicate? Does it have to be a Boolean?

(2) Are both the true and false branches evaluated?

(3) Are **if**-conditionals values or expressions?

(4) Where can **if** occur? (hint: revisit Lecture 3's first-class function definition for some contexts to explore)

Provide a table of observations that you have made about the behavior of **if** in different program contexts, **as well as answers to the questions above**.

Part 2: Practicing Racket (70 points)

1. string-multiply (10 points)

Write a function that, given a string s and a number n, returns a string consisting of s repeated n times. Call this **string-multiply**.

For example:

> (string-multiply "cat" 5)

catcatcatcatcat

2. string-alphabetized? (10 points)

Write a function that checks whether the characters in a string are in alphabetical order. If so, return true; if not return false. Call it **string-alphabetized?**.

For example:

> (string-alphabetized? "act")

#t

> (string-alphabetized? "cat")

#f

3. string-join (10 points)

Write a function that, when given a list of strings and a string s, returns a string consisting of all of the strings in the list, in order, separated by s. Call this **string-join**.

For example:

> (string-join (list "hello" "world") " ")

"hello world"

4. lst-alphabetized? (10 points)

Write a function that checks whether the strings in a list are in alphabetical order. If so, return true; if not return false. Call this **lst-alphabetized?**.

For example:

> (lst-alphabetized? (list "apple" "banana" "carrot"))

#t

5. pyramid-print (15 points)

A. Write a function that takes a string s and a number n as arguments and returns a string that, when printed, displays an ascii art pyramid with height n, where each row contains two more instances of the character s than the previous. Call this function **pyramid-print**.

For instance, for input char = x and n = 4, the returned string prints as:

x
xxx
xxxxx
xxxxxxx

B. Make a new version of your pyramid-print function that prints a centered pyramid. Call this **centered-pyramid-print**.

6. sort-students (15 points)

Write a function that when given a list of student-year pairs, sorts them by class year. Because we have not learned any data structures other than lists, you should return **a list containing 4 lists (1 for each class year).**

For instance, given the list ((Clara 2021) (Indranie 2023) (Ida 2023)), your program should return:

```
> (sort-students (Clara 2021) (Indranie 2023) (Ida 2023)))

'((Clara) () (Indranie Ida) ()).
```

(It's easier for your program to always return 4 lists, so return an empty list when there are no students from a particular year.)

Hint: you may want to define some helper functions to make this easier!

Extra Credit

1. centered-print

Write a function called **centered-print** that takes a list of strings and prints them one per line, centered. The longest string should have no padding on the left.

2. mutually recursive functions

Tail-recursion is more efficient because there is no need to keep track of the recursive stack, since all of the computation is contained within the recursive call.

The following code shows two **mutually recursive functions**.

```
(define (is-even n)
                  (if (= n 0)
                      #t
                      (is-odd (- n 1))))

(define (is-odd n)
                 (if (= n 0)
                      #f
                      (is-even (- n 1))))
```

Do **is-even** and **is-odd** have the same efficiency advantage as tail-recursive functions? Trace through the evaluation of a couple of examples to reach your conclusion.