Fall 2022
Becker, Kamal, Nasr Esfahani, Tompkins, Van Leeuwen, Vasiga

Assignment: 9

Due:	Tuesday, November 29, 2022 9:00 pm
Coverage:	End of Module 16
Language level:	Intermediate Student with Lambda
Allowed recursion:	No explicit recursion (see note below)
Files to submit:	`q02.rkt, bw-images.rkt, funfun.rkt`

- Make sure you read the A09 Official Post and FAQ post on Piazza for the answers to frequently asked questions.
- Your solutions for assignments will be graded on both correctness and on readability. This assignment will be auto-graded and hand-marked.
- Policies from Assignment A08 carry forward.
- Submit examples for all questions by **Friday, November 25 at 8:00AM** (before the due date of assignment A09). See the Example Submissions and Basic Test Results post on Piazza for examples of correct and incorrect example submissions.
- **You may not use explicit recursion for any question.** This implies that functions which involve an application of themselves, either directly or via mutual recursion, are not allowed.
- If you need a helper function, it must be defined in a `local` or using `lambda`. Note that local helper functions cannot use explicit recursion either.
- You should use the higher-order functions that are listed in DrRacket under Help→ Help Desk → How to Design Programs Languages → 4.18 Higher-Order Functions.
- For this assignment you are not allowed to use the built-in Racket functions `member?`, `reverse`, `append` or any of the built-in string functions (e.g. `string-append`, `substring`, `string-fill`, etc.). You are only allowed to use `string->list` or `list->string` where needed, and you may also use `list-ref`.

Here are the assignment questions that you need to submit.

1. In this question you will perform step-by-step evaluations of Racket programs, as you did in A01. Please review the instructions on stepping in A01 and complete the following:

   (a) The two required problems in "Module 13: Lambdas" and

   (b) The two required problems in "Module 14: Abstract List Functions"

   on the CS135 Stepping Practice website.

2. (a) Write the function `alphanumeric-only` that consumes a list of strings and produces the same list of strings keeping only the alphanumeric strings. Here are a few examples:

```
(check-expect (alphanumeric-only '("5he%llo" "12!3hu")) '())
(check-expect (alphanumeric-only '("5hello" "123hu" "89huyt#"))
                                 '("5hello" "123hu"))
(check-expect (alphanumeric-only '("hello" "?u" "123hu"))
                                 '("hello" "123hu"))
```

   Hint: You are allowed to use builtin Racket functions `char-alphabetic?` and `char-numeric?`.

   (b) Write the function `remove-outliers` that consumes a non-empty list of numbers, and produces a list of the same numbers in the same order, excluding the outliers. Let the list of numbers be $x_1, x_2, x_3, \ldots, x_n$, then the mean μ and the standard deviation σ are computed as follows:

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n}}$$

   Then, we will define an outlier as any number x where $x < (mean - \sigma)$ or $x > (mean + \sigma)$.

   Here are a few examples:

```
(check-expect (remove-outliers '(1)) '(1))
(check-expect (remove-outliers '(9 8 -3 -9 -8 0 0 0 0 3))
                               '(-3 0 0 0 0 3))
```

   (c) Write the function `zero-fill`, which consumes a string no longer than 20 characters. It produces the same string but with zeros added to the end, as necessary, so that the string is exactly 20 characters long.

   This problem is taken from computer networks where the string represents a "datagram". For the network to work efficiently, all the datagrams must be the same length – in our case, 20 characters.

   For this question, you are not allowed to use any built-in string functions, except `list->string` and `string->list`. Here are a few examples for `zero-fill`:

```
(check-expect (zero-fill "abcdefghijklmn") "abcdefghijklmn000000")
(check-expect (zero-fill "he00llo") "he00llo0000000000000")
```

(d) Write the function `remove-duplicates` which consumes a list of numbers and produces a list containing only the first occurrence of each number, in the order they occur.

For example:

```
(check-expect
     (remove-duplicates (list 1 2 3 4 3 2 4 1)) (list 1 2 3 4))
```

Place all your functions for this question in the file `q02.rkt`.

3. This question focusses on simple black and white 2-dimensional images. We will represent a 2D black and white image as a list of lists of 0's (white) and 1's (black). We will define a black and white 2D image as:

```
;; BW-Pixel is (anyof 0 1)
;; 2D-Image is (listof (ne-listof BW-Pixel))
;;   Requires: inner lists of 2D-Image are of same length
```

Let us define a 2D black and white image, `image-L`, as follows:

```
(define image-L '((1 0 0 0)
                  (1 0 0 0)
                  (1 0 0 0)
                  (1 1 1 1)))
```

Then, the reflection of `image-L` across the x-axis, that is, the horizontal number line in the Cartesian coordinate system is:

```
(define image-L-reflect-x '((1 1 1 1)
                            (1 0 0 0)
                            (1 0 0 0)
                            (1 0 0 0)))
```

And the reflection of `image-L` across the y-axis, that is, the vertical number line in the Cartesian coordinate system is:

```
(define image-L-reflect-y '((0 0 0 1)
                            (0 0 0 1)
                            (0 0 0 1)
                            (1 1 1 1)))
```

(a) Write the function `(invert image)` that consumes a `2D-Image image`, and produces a `2D-Image` that is the inverted, such that, the black pixels are white and the white pixels are black. For example:

```
(define image-L-inverted '((0 1 1 1)
                           (0 1 1 1)
                           (0 1 1 1)
                           (0 0 0 0)))
 (check-expect (invert image-L) image-L-inverted)
```

(b) Write the function `reflect-x-axis` that consumes a `2D-Image` and produces a `2D-Image` that represents the reflection of the `2D-Image` across the x-axis. For example,

```
(check-expect (reflect-x-axis image-L) image-L-reflect-x)
```

(c) Write the function `reflect-y-axis` that consumes a `2D-Image` and produces a `2D-Image` that represents the reflection of the `2D-Image` across the y-axis. For example,

```
(check-expect (reflect-y-axis image-L) image-L-reflect-y)
```

(d) Write the function `transpose` which consumes a `2D-Image` and produces a `2D-Image` that represents the transposed image. To transpose an image, we exchange the rows and columns: the first row becomes the first column, the second row becomes the second column, and so on. For example:

```
(check-expect (transpose image-L)
              '((1 1 1 1)
                (0 0 0 1)
                (0 0 0 1)
                (0 0 0 1)))
(check-expect (transpose '((1 0 0 1 1)))
              '((1) (0) (0) (1) (1)))
```

Place all your functions for this question in the file `bw-images.rkt`.

4. Solve each of the following functions dealing with functions.

(a) Write the function `multi-apply` which consumes a list of functions, all of which consume and produce the same type, and a value, of that same type. This function applies the functions from left to right on the given value. That is, if the list of functions is `(f_1 f_2 ... f_n)`, and the value is *v*, `multi-apply` will produce `(f_n(...(f_2(f_1(v))) ...)`. For example:

```
(check-expect (multi-apply (list sub1 sqr add1) 3) 5)
```

(b) Write a function `aop` which consumes a natural number `n` and produces a function that behaves like the "all one polynomial" of degree n, which is a polynomial where all the coefficients are one. Mathematically, if $n = 2$, the function `aop` would produce a function equivalent to $x^2 + x + 1$. For example:

```
(check-expect ((aop 4) 2) 31)  ;; 1+2+4+8+16
(check-expect ((aop 0) 5) 1)   ;; 1
(check-expect ((aop 16) 1) 17)
```

(c) Write a function `multi-compose` which consumes a list of functions, all of which consume one value of the same type, and produce a value of the same type. The function `multi-compose` produces a function which consumes one value, and will compose the functions from right to left on that value. That is, if the list of functions `(f_1 f_2 ... f_n)`, the function `multi-compose` will produce a function equivalent to `(f_1(f_2(...(f_n(x))...)))`. For example:

```
(check-expect ((multi-compose (list sub1 sqr add1)) 3) 15)
```

Place all of your functions for this question in the file `funfun.rkt`.

This concludes the list of questions for which you need to submit solutions.
Remember to always check your email for the basic test results after making a submission.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Professor Temple does not trust the built-in functions in Racket. In fact, Professor Temple does not trust constants, either. Here is the grammar for the programs Professor Temple trusts.

⟨exp⟩ = ⟨var⟩|(**lambda** (⟨var⟩) ⟨exp⟩) | (⟨exp⟩⟨exp⟩)

Although Professor Temple does not trust **define**, we can use it ourselves as a shorthand for describing particular expressions constructed using this grammar.

It doesn't look as if Professor Temple believes in functions with more than one argument, but in fact Professor Temple is fine with this concept; it's just expressed in a different way. We can create a function with two arguments in the above grammar by creating a function which consumes the first argument and returns a function which, when applied to the second argument, returns the answer we want. This generalizes to multiple arguments.

But what can Professor Temple do without constants? Quite a lot, actually. To start with, here is Professor Temple's definition of zero. It is the function which ignores its argument and returns the identity function.

```
(define my-zero (lambda (f) (lambda (x) x)))
```

Another way of describing this representation of zero is that it is the function which takes a function f as its argument and returns a function which applies f to its argument zero times. Then "one" would be the function which takes a function f as its argument and returns a function which applies f to its argument once.

```
(define my-one (lambda (f) (lambda (x) (f x))))
```

Work out the definition of "two". How might Professor Temple define the function `add1`? Show that your definition of `add1` applied to the above representation of zero yields one, and applied to one yields two. Can you give a definition of the function which performs addition on its two arguments in this representation? What about multiplication?

Now we see that Professor Temple's representation can handle natural numbers. Can Professor Temple handle Boolean values? Sure. Here are Professor Temple's definitions of true and false.

```
(define my-true (lambda (x) (lambda (y) x)))
(define my-false (lambda (x) (lambda (y) y)))
```

Show that the expression `((c a) b)`, where `c` is one of the values `my-true` or `my-false` defined above, evaluates to `a` and `b`, respectively. Use this idea to define the functions `my-and`, `my-or`, and `my-not`.

What about `my-cons`, `my-first`, and `my-rest`? We can define the value of `my-cons` to be the function which, when applied to `my-true`, returns the first argument `my-cons` was called with, and when applied to the argument `my-false`, returns the second. Give precise definitions of `my-cons`, `my-first`, and `my-rest`, and verify that they satisfy the algebraic equations that the regular Racket versions do. What should `my-empty` be?

The function `my-sub1` is quite tricky. What we need to do is create the pair $(0,0)$ by using `my-cons`. Then we consider the operation on such a pair of taking the "rest" and making it the "first", and making the "rest" be the old "rest" plus one (which we know how to do). So the tuple $(0,0)$ becomes $(0,1)$, then $(1,2)$, and so on. If we repeat this operation n times, we get $(n-1,n)$. We can then pick out the "first" of this tuple to be $n-1$. Since our representation of n has something to do with repeating things n times, this gives us a way of defining `my-sub1`. Make this more precise, and then figure out `my-zero?`.

If we don't have **`define`**, how can we do recursion, which we use in just about every function involving lists and many involving natural numbers? It is still possible, but this is beyond even the scope of this challenge; it involves a very ingenious (and difficult to understand) construction called the Y combinator. Here are a few reading resources on the Y combinator

- University of Toronto lecturer David Liu has an explanation of the Y combinator,

- Chapter 9 has an example of the Y combinator, and
- medium.com also has a discussion on the Y combinator.

Be warned that this is truly mindbending.

Professor Temple has been possessed by the spirit of Alonzo Church (1903–1995), who used this idea to define a model of computation based on the definition of functions and nothing else. This is called the lambda calculus, and he used it in 1936 to show a function which was definable but not computable (whether two lambda calculus expressions define the same function). Alan Turing later gave a simpler proof which we discussed in the enhancement to Assignment 7. The lambda calculus was the inspiration for LISP, the predecessor of Racket, and is the reason that the teaching languages retain the keyword `lambda` for use in defining anonymous functions.