CSC 330
Programming Languages
Assignment 4

Note 1 **This assignment is to be done individually**

Note 2 You can discuss the assignment with others, but sharing and copying code is prohibited.

A note on Academic Integrity and Plagiarim

Please review the following documents:

- Standards for Professional Behaviour, Faculty of Engineering:
  `https://www.uvic.ca/engineering/assets/docs/professional-behaviour.pdf`
- Policies Academic Integrity, UVic:
  `https://www.uvic.ca/students/academics/academic-integrity/`
- Uvic's Calendar section on Plagirism:
  `https://www.uvic.ca/calendar/undergrad/index.php#/policy/Sk_0xsM_V`

Note specifically:

Plagiarism
Single or multiple instances of inadequate attribution of sources should result in a failing grade for the work. A largely or fully plagiarized piece of work should result in a grade of F for the course.

You are responsible for your own submission, but you could also be responsible if somebody plagiarizes your submission.

Objectives

After completing this assignment, you will have:

- experience programming in Racket.
- experience using thunks and lazy evaluation.
- experience with memoization.

Introduction

Your task, should you choose to accept it

Part 1

1. Write a function `add-pointwise` that takes two lists of lists of numbers and returns its pointwise addition. The lists might have different length. Return a list of length equal to the length of the longest list. The i-th element of the list is equal to the sum of the i-th elements of each list (in the case of a list being shorter, use zero for the i-th value of such list). Call `(error "illegal parameter")` if any of the parameters is not a list. For example:

```
>  (add-pointwise '(1 2 3) '(4 2))
'(5 4 3)
```

2. Write a function `add-pointwise-lists` that takes a list of lists of integers as its parameter and returns the pointwise addition of these lists. This function is a generalization of `add-pointwise`. Call `(error "illegal parameter")` if the parameter is invalid. **Restriction: this function must be recursive**.

   For example:

   ```
   > (add-pointwise-lists '((1 1) (2 2 2 2) (3) ()))
   '(6 3 2 2)
   ```

3. Write a function `add-point-wise-lists-2` that is programming language equivalent to `pair-with-itself`. **Restriction: do not use recursion**. You can use `add-pointwise`, however.

4. Write a function `stream-for-n-steps` that takes a stream `s` and a number `n`. It returns a list holding the first `n` values produced by `s` in order. Note: You can use this function to tests your streams. For example, assume that the stream `nat-num-stream` returns the natural numbers (starting at 0):

   ```
   > (stream-for-n-steps nat-num-stream 10)
   '(0 1 2 3 4 5 6 7 8 9)
   ```

5. Write a stream called `fibo-stream` that corresponds to the Fibonacci sequence. Hint: the first value needs to be treated differently and the thunk should receive two parameters.

   ```
   > (stream-for-n-steps fibo-stream 10)
   '(0 1 1 2 3 5 8 13 21 34)
   ```

6. Write a function called `filter-stream` that takes two parameters (the first is a function, the second a stream) and returns a stream. The function returns a boolean that determines if a given value returned by the stream is to be kept or not (`filter-stream` is similar to `filter`, but instead of a list, it is *filtering* a stream). Example:

   ```
   > (stream-for-n-steps
           (filter-stream (lambda (i) (> i 5)) nat-num-stream) 5)
   '(6 7 8 9 10)
   ```

7. Using `filter-stream` and `nat-num-stream` write a stream called `palyndromic-numbers`. A palyndromic number is one that is the same when its digits (in base 10) are reversed. Hint: convert the number to a string, then the string to a list of characters. You can compare two lists using `equal?`

   ```
   > (stream-for-n-steps palyndromic-numbers 20)
   '(0 1 2 3 4 5 6 7 8 9 11 22 33 44 55 66 77 88 99 101)
   ```

8. There exists a family of numeric sequences that follow a simple pattern:

$$f(i_0), f(i_1), f(i_2), f(i_3)...$$

where

$$i_j = i_{j-1} + \delta$$

Define a macro called `create-stream` that will create such stream that takes 3 parameters: `f`, `i0` and `delta`. The integer `i0` corresponds to i_0

The syntax of the macro should be:

```
(create-stream name using f starting at i0 with increment delta)
```

This will create a stream called name using `f` as its function, with starting value `i0` and increment `delta`.

- `i0` should be evaluated only once when the stream is first used to generate the first value.
- `delta` should be evaluated once every time the stream is used (except in the first invocation).

Example:

```
> (create-stream squares using (lambda (x) (* x x))
           starting at  (begin (print "starting") 5)
           with increment (begin (print "inc") 2))
> (squares)
"starting"'(25 . #<procedure:...ssign/assign.rkt:112:32>)
> (stream-for-n-steps squares 5)
"starting""inc""inc""inc""inc"'(25 49 81 121 169)
```

Part 2

1. Write a function `vector-assoc` that takes a value `v` and a vector `vec`. It should behave like Racket's `assoc` library function except:

   (a) it processes a vector (Racket's name for an array) instead of a list and

   (b) it allows vector elements not to be pairs in which case it skips them.

   Process the vector elements in order starting from 0. Use library functions `vector-length`, `vector-ref`, and `equal?`. Return `#f` if no vector element is a pair with a `car` field equal to `v`, else return the first pair with an equal `car` field.

2. Write a function `cached-assoc` that takes a list `xs` and a number `n` and returns a function that takes one argument `v` and returns the same thing that `(assoc n xs)` would return. However, you should use an n-element *cache of recent results* to possibly make this function faster than just calling `assoc` (if `xs` is long and a few elements are returned often). The cache should be a vector of length `n` that is

created by the call to `cached-assoc` and used-and-possibly-mutated each time the function returned by cached-assoc is called (use `vertor-assoc` to search the vector).

The cache starts empty (all elements `#f`). When the function returned by `cached-assoc` is called, it first checks the cache for the answer. If it is not there, it uses `assoc` and `xs` to get the answer and if the result is not `#f` (i.e., `xs` has a pair that matches), it adds the pair to the cache before returning (using `vector-set!`). The cache slots are used in a round-robin fashion: the first time a pair is added to the cache it is put in position 0, the next pair is put in position 1, etc. up to position `n - 1` and then back to position 0 (replacing the pair already there), then position 1, etc.

Hints:

- In addition to a variable for holding the vector whose contents you mutate with `vector-set!`, use a second variable to keep track of which cache slot will be replaced next. After modifying the cache, increment this variable (with `set!`) or set it back to 0 (as needed).
- To test your cache, it can be useful to add print expressions so you know when you are using the cache and when you are not. But remove these print expressions before submitting your code.

You must use mutation in this function.

Tests provided

The file `main.sml` contains some tests that will clarify the behaviour of your functions. Run this program from the command line:

```
racket main.sml
```

I recommend you modify this file to add your own tests. **You will not submit this file**. Make sure you test your submission with an unmodified version of this file to make sure your program works as expected. You can exchange tests with other students (but you are forbidden from sharing the solutions to the assignment).

Restrictions

- Some functions have specific restrictions stated in along their description.
- You cannot use mutation in any function, except `cached_assoc`.
- Unless specified in the description of a function, you are not expected to check the types of the parameters of the functions (i.e. the functionality of the function is undefined if a parameter of the wrong type is passed).

Evaluation

Solutions should:

1. Run in `linux.csc.uvic.ca` If a program does not compile/run **for any reason** it will receive a zero.
2. Satisfy the restrictions above. You will lose marks if you don not follow them.
3. Pass tests. Your grade will be determined based on how many tests it passes.

Second submission

If you choose, you can submit this assignment a second time (the deadline will be set after the results of the first submission are published). The final grade of your assignment will be:

$$max(first\_submission, 0.5 * second\_submission)$$

You are not required to submit a second time.

What to submit

1. Using conneX, submit your version of the file `functions.rkt`. **Do not submit any other files**.
2. You should assume that your submission was received correctly only if both of the following two conditions hold:
   (a) You receive a confirmation email from conneX.
   (b) You can download and view your submission, and it contains the correct data.
3. In the event that your submission is not received by the marker, you will need to demonstrate that both of the above conditions were met if you want to argue that there was a submission error.