Programming Languages (Coursera / University of Washington)
Assignment 4

You will write 10 Racket functions (not counting helper functions). Writing a Racket macro is a challenge problem.

Download several files from the course website and put them all in the same directory. Add to `hw4.rkt` to complete your homework.

Provided Code for Graphical Output:

The code at the top of `hw4combined_tests_with_graphics.rkt` uses a graphics library to provide a simple, entertaining (?) outlet for your streams. You need not understand this code (though it is not complicated) or even use it, but it may make the homework more fun. This is how you use it:

- `(open-window)` returns a graphics window you can pass as the first argument to `place-repeatedly`.
- `(place-repeatedly window pause stream n)` uses the first `n` values produced by `stream`. Each stream element must be a pair where the first value is an integer between 0 and 5 inclusive and the second value is a string that is the name of an image file (e.g., `.jpg`). (Sample image files that will work well are available on the course website. Put them in the same directory as your code.) Every `pause` seconds (where `pause` is a decimal, i.e., floating-point, number), the next stream value is retrieved, the corresponding image file is opened, and it is placed in the window using the number in the pair to choose its position in a 2x3 grid as follows:

0	1	2
3	4	5

Two of the provided tests demonstrate how to use `place-repeatedly`. The provided tests require you to complete several of the problems, of course. We hope these tests' expected (visual) behavior is not difficult for you to figure out.

Small Example Tests:

The example tests in `hw4test.rkt` are grouped into a small test suite using Racket's unit-testing framework. You do not need to understand the details, but it is worthwhile to do so by reading `http://docs.racket-lang.org/rackunit/quick-start.html`.

Helpful Guide / Warning:

The first three problems are "warm-up" exercises for Racket. Subsequent problems dive into streams (4–8) and memoization (10). Some short problems may be difficult. Go slowly and focus on using what you learned about thunks, streams, etc.

Some problems require that you use a few standard-library functions that were not used in lecture. See the Racket documentation at `http://docs.racket-lang.org/`, particularly The Racket Guide, as necessary — looking up library functions even in languages new to you is an important skill. It is fine to discuss with others in the class what library functions are useful and how they work.

Turn-in Instructions (same as in Part A of the course):

First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.

Problems:

1. Write a function `sequence` that takes 3 arguments `low`, `high`, and `stride`, all assumed to be numbers. Further assume `stride` is positive. `sequence` produces a list of numbers from `low` to `high` (including `low` and possibly `high`) separated by `stride` and in sorted order. Sample solution: 4 lines. Examples:

Call	Result
`(sequence 3 11 2)`	`'(3 5 7 9 11)`
`(sequence 3 8 3)`	`'(3 6)`
`(sequence 3 2 1)`	`'()`

2. Write a function `string-append-map` that takes a list of strings `xs` and a string `suffix` and returns a list of strings. Each element of the output should be the corresponding element of the input appended with `suffix` (with no extra space between the element and `suffix`). You must use Racket-library functions `map` and `string-append`. Sample solution: 2 lines.

3. Write a function `list-nth-mod` that takes a list `xs` and a number `n`. If the number is negative, terminate the computation with `(error "list-nth-mod: negative number")`. Else if the list is empty, terminate the computation with `(error "list-nth-mod: empty list")`. Else return the i^{th} element of the list where we *count from zero* and i is the remainder produced when dividing `n` by the list's length. Library functions `length`, `remainder`, `car`, and `list-tail` are all useful – see the Racket documentation. Sample solution is 6 lines.

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. Assume `n` is non-negative. Sample solution: 5 lines. Note: You can test your streams with this function instead of the graphics code.

5. Write a stream `funny-number-stream` that is like the stream of natural numbers (i.e., 1, 2, 3, ...) except numbers divisble by 5 are negated (i.e., 1, 2, 3, 4, -5, 6, 7, 8, 9, -10, 11, ...). Remember a stream is a thunk that when called produces a pair. Here the car of the pair will be a number and the cdr will be another stream.

6. Write a stream `dan-then-dog`, where the elements of the stream alternate between the strings `"dan.jpg"` and `"dog.jpg"` (starting with `"dan.jpg"`). More specifically, `dan-then-dog` should be a thunk that when called produces a pair of `"dan.jpg"` and a thunk that when called produces a pair of `"dog.jpg"` and a thunk that when called... etc. Sample solution: 4 lines.

7. Write a function `stream-add-zero` that takes a stream `s` and returns another stream. If `s` would produce v for its i^{th} element, then `(stream-add-zero s)` would produce the pair `(0 .` v`)` for its i^{th} element. Sample solution: 4 lines. Hint: Use a thunk that when called uses `s` and recursion. Note: One of the provided tests in the file using graphics uses `(stream-add-zero dan-then-dog)` with `place-repeatedly`.

8. Write a function `cycle-lists` that takes two lists `xs` and `ys` and returns a stream. The lists may or may not be the same length, but assume they are both non-empty. The elements produced by the stream are pairs where the first part is from `xs` and the second part is from `ys`. The stream cycles forever through the lists. For example, if `xs` is `'(1 2 3)` and `ys` is `'("a" "b")`, then the stream would produce, `(1 . "a")`, `(2 . "b")`, `(3 . "a")`, `(1 . "b")`, `(2 . "a")`, `(3 . "b")`, `(1 . "a")`, `(2 . "b")`, etc.

   Sample solution is 6 lines and is more complicated than the previous stream problems. Hints: Use one of the functions you wrote earlier. Use a recursive helper function that takes a number `n` and calls itself with `(+ n 1)` inside a thunk.

9. Write a function `vector-assoc` that takes a value `v` and a vector `vec`. It should behave like Racket's `assoc` library function except (1) it processes a vector (Racket's name for an array) instead of a list, (2) it allows vector elements not to be pairs in which case it skips them, and (3) it always takes exactly two arguments. Process the vector elements in order starting from 0. You must 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. Sample solution is 9 lines, using one local recursive helper function.

10. 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 v 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 must be a Racket vector of length n that is created by the call to `cached-assoc` (use Racket library function `vector` or `make-vector`) and used-and-possibly-mutated each time the function returned by `cached-assoc` is called. Assume n is positive.

    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.
    - 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.
    - Sample solution is 15 lines.

11. **(Challenge Problem:)** Define a macro that is used like `(while-less e1 do e2)` where `e1` and `e2` are expressions and `while-less` and `do` are syntax (keywords). The macro should do the following:

    - It evaluates `e1` exactly once.
    - It evaluates `e2` at least once.
    - It keeps evaluating `e2` until and only until the result is not a number less than the result of the evaluation of `e1`.
    - Assuming evaluation terminates, the result is `#t`.
    - Assume `e1` and `e2` produce numbers; your macro can do anything or fail mysteriously otherwise.

    Hint: Define and use a recursive thunk. Sample solution is 9 lines. Example:

    ```
    (define a 2)
    (while-less 7 do (begin (set! a (+ a 1)) (print "x") a))
    (while-less 7 do (begin (set! a (+ a 1)) (print "x") a))
    ```

    Evaluating the second line will print `"x"` 5 times and change `a` to be 7. So evaluating the third line will print `"x"` 1 time and change `a` to be 8.

Assessment: We will automatically test your functions on a variety of inputs, including edge cases. We may automatically check that you used required library functions. We will also ask peers to evaluate your code for simplicity, conciseness, elegance, and good formatting including indentation and line breaks. Do not use mutation except in problem 10. (If doing the challenge problem, you will also use mutation to *test* problem 11.)