CSE 341, Fall 2011, Assignment 4
Due: Wednesday November 9, 11:00PM

You will write 9 Racket functions (not counting helper functions) and 1 Racket macro.

Download `hw4provided.rkt` and `hw4providedTests.rkt` from the course website. Add to these files (and rename them) to complete your homework.

Provided Code:

The code at the top of `hw4providedTests.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 is 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.) 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

One of the provided tests provides an example use of `place-repeatedly`. This code requires you to complete several of the problems, of course. You should be able to figure out how this testing code should behave.

Warning:

The first three problems are "warm-up" exercises for Racket. Subsequent problems dive into streams (4–7), memoization (9), and macros (10). Some short problems may be difficult.

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. For all i, the i^{th} element of the output should be the concatenation of the i^{th} element of `xs` and `suffix`. Use library functions `map` and `string-append` (see the Racket documentation as necessary). Sample solution: 2 lines.

3. Write a function `list-nth-mod` that takes a list and a number. If the number is negative or the list is empty, use `error` to print an appropriate error message. (Write `(error "blah")` to terminate computation with message `"blah"`.) Otherwise, 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. Sample solution is 6 lines.

4. Write a stream `curry-then-steele`, where the elements of the stream alternate between the strings `"curry.jpg"` and `"steele.jpg"` (starting with `"curry.jpg"`). More specifically, `curry-then-steele` should be a thunk that when called produces a pair of `"curry.jpg"` and a thunk that when called produces a pair of `"steele.jpg"` and a thunk that when called... etc. Hint: Use 2 mutually recursive functions. Sample solution: 4 lines.

5. 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: You can test `(stream-add-zero curry-then-steele)` with `place-repeatedly`.

6. 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. Sample solution: 5 lines. Note: You can test your streams with this function instead of the graphics code.

7. 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. 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.

8. 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 and (2) 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. Sample solution is 9 lines, using one local recursive helper function.

9. 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`. 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.

   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.
   - Sample solution is 15 lines.

10. 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` one or more times.
    - If the result of evaluating `e2` is a number less than the result of evaluating `e1`, then `e2` should be evaluated again, otherwise the macro finishes, returning `#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.

11. **Challenge Problem:** Write `cycle-lists-challenge`. It should be equivalent to `cycle-lists`, but its implementation must be more efficient. In particular, for each time the stream produces a new value, the code must perform only two `car` operations and two `cdr` operations, including operations performed by any function calls. So, for example, you cannot use `length` because it uses `cdr` multiple times to compute a list's length.

12. **Challenge Problem:** Write `cached-assoc-lru`, which is like `cached-assoc` except it uses a policy of "least recently used" for deciding which cache slot to replace. That is, when replacing a pair in the cache, you must choose the pair that was least recently returned as an answer. Doing so requires maintaining extra state.

Test your functions: Put your testing code in a second file. We will not grade it, but you must turn it in.

Assessment: Your solutions should be correct, in good style, and use only features we have used in class. In particular, *only problems 9 and 12 should use mutation.* (You will also use mutation to *test* problem 10.)

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw4.rkt`, where `lastname` is replaced with your last name. Put tests in `lastname_hw4_test.rkt`.
- Turn in your files using the Catalyst dropbox link on the course website.