CSE 341, Spring 2020, Assignment 4
Due: Tuesday, May 19, 2020, 11:59PM

Overview

You will write 10 Racket functions (not counting helper functions) and 1 Racket macro. The first three problems are "warm-up" exercises for Racket. Subsequent problems dive into streams (4–8), memoization (9–10), and macros (11). Most problems have short descriptions and short solutions, but short problems may be difficult. Go slowly and focus on using what we've learned in class 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.

Download `hw4.zip` from the course website which includes the files `hw4.rkt` and `hw4tests.rkt`. Add to these files to complete your assignment.

Provided Test Code:

The code at the top of `hw4tests.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.

Problems:

The first three problems will give you some warm-up practice with Racket.

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

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

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.

In the next four problems, you will consume, produce, and manipulate streams.

4. Write a function `stream-for-k-steps` that takes a stream `s` and a number `k`. It returns a list holding the first `k` values produced by `s` in order. Assume `k` 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 6 are negated (i.e., 1, 2, 3, 4, 5, -6, 7, 8, 9, 10, 11, -12, 13, ...). 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-one` that takes a stream `s` and returns another stream. If `s` would produce v for its i^{th} element, then `(stream-add-one s)` would produce the pair `(1 .` 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 uses `(stream-add-one 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.

The next two problems have you implement your own version of `assoc` that uses caching (via memoization) to improve performance.

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 `caching-assoc` that takes a list `xs` and a positive 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 should be a vector of length n that is created by the call to `caching-assoc` and used-and-possibly-mutated each time the function returned by `caching-assoc` is called.

    The cache starts empty (all elements `#f`). When the function returned by `caching-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.

Finally, you will write a Racket macro.

11. Define a macro that is used like `(while-greater e1 do e2)` where `e1` and `e2` are expressions and `while-greater` 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 greater 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 7)
    (while-greater 2 do (begin (set! a (- a 1)) (print "x") a))
    (while-greater 2 do (begin (set! a (- a 1)) (print "x") a))
    ```

    Evaluating the second line will print `"x"` 5 times and change `a` to be 2. So evaluating the third line will print `"x"` 1 time and change `a` to be 1.

---

12. **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.

13. **Challenge Problem:** Write `caching-assoc-lru`, which is like `caching-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.

Testing

In addition to implementing the functions described above, you must write a suite of tests to verify that your functions work correctly. Tests should be written using the approach shown in section, and should be comprehensive enough to fully verify that your functions work as indicated. Be sure to consider edge cases and unusual (but valid) inputs. Truly exceptional test suites may receive a small amount of extra credit.

Assessment

To receive full credit, your solutions should be:

- Functionally correct
- Written in good style according to the style guide, including indentation and line breaks
- Written using only features discussed in class through Wednesday, May 13 (Lecture 15).
- Written **without** using mutation, except for problems 10 and 13. (You will also need mutation to test problem 11.)
- Written **without** using Racket's built-in streams. (These do not work the same way we are using streams.)

Turn-in Instructions

- Add all your solutions to the main problems and challenge problems (if you worked on them) to the file `hw4.rkt`.
- Add all your tests to the file `hw4tests.rkt`.
- Follow the link on the course website to submit your files to Gradescope.
- The Gradescope autograder will confirm that you have submitted the correct file and that your code compiles. **Submissions that do not compile will receive a 15% penalty!**