Login

Assignment 1: Recursion and Higher-Order Functional Abstractions

Recursion is the root of computation since it trades description for time.

Guidelines for this assignment

- Solutions must be recursive (**naturally recursive** unless not possible), or credit will not be given.
- You may not use built-in procedures that handle the bulk of the work.
- Please feel free to submit your work as many times as you wish, we will grade the last submission prior to the due date (and time).
- Named lets (aka "let loop") should not be used on this assignment.
- The objective is not simply to write programs that get the correct answers; it is to write answers in the style of programs written in class.
- **Make sure to title your file `a1.rkt` when you submit your homework.**

Testing your assignment

You must test your solutions before submitting your assignment. We have provided a suite of test cases to get you started. To run these tests, you must download the a1-student-tests.rkt test file. To use these tools, do the following in a Racket REPL (the code and test suite must be located in the same directory):

```
> (require "a1-student-tests.rkt")
> (test-file #:file-name "a1.rkt")
...
```

and that should get you going. Of course, **these tests are not exhaustive; you should add your own tests as well**.

Note

As you proceed with this assignment, you may find the following resources helpful.

- Notes on recursive functions for repeated addition/multiplication/exponentiation/etc.
- Simplification of the Ackermann function to standard form.

Assignment

Write the following recursive Racket procedures. Place all of your code in a file named `a1.rkt`, and submit it via Canvas [http://canvas.iu.edu]. Please make sure your file has exactly this filename, and that it runs, before submitting.

0. **We've recently updated the course policies for the semester. Please read through them before beginning the rest of the assignment.**

1. Define and test a procedure `countdown` that takes a natural number and returns a list of the natural numbers less than or equal to that number, in descending order.

```
> (countdown 5)
(5 4 3 2 1 0)
```

2. Define and test a procedure `insertR` that takes two symbols and a list and returns a new list with the second symbol inserted after each occurrence of the first symbol. **For this and later questions, these functions need only hold over `eqv?`-comparable structures.**

```
> (insertR 'x 'y '(x z z x y x))
(x y z z x y y x y)
```

3. Define and test a procedure `remv-1st` that takes a a symbol and a list and returns a new list with the first occurrence of the symbol removed.

```
> (remv-1st 'x '(x y z x))
(y z x)
> (remv-1st 'y '(x y z y x))
(x z y x)
```

4. Define and test a procedure `list-index-ofv?` that takes an element and a list and returns the (base 0) index of that element in the list. A list missing that element will be considered bad data.

```
> (list-index-ofv? 'x '(x y z x x))
0
> (list-index-ofv? 'x '(y z x x))
2
```

5. Define and test a procedure `filter` that takes a predicate and a list and returns a new list containing the elements that satisfy the predicate. A *predicate* is a procedure that takes a single argument and returns either `#t` or `#f`. The `number?` predicate, for example, returns `#t` if its argument is a number and `#f` otherwise. The argument satisfies the predicate, then, if the predicate returns `#t` for that argument.

```
> (filter even? '(1 2 3 4 5 6))
(2 4 6)
```

6. Define and test a procedure `zip` that takes two lists and forms a new list, each element of which is a pair formed by combining the corresponding elements of the two input lists. If the two lists are of uneven length, then drop the tail of the longer one.

```
> (zip '(1 2 3) '(a b c))
((1 . a) (2 . b) (3 . c))
> (zip '(1 2 3 4 5 6) '(a b c))
((1 . a) (2 . b) (3 . c))
> (zip '(1 2 3) '(a b c d e f))
((1 . a) (2 . b) (3 . c))
```

7. Define and test a procedure `map` that takes a procedure `p` of one argument and a list `ls` and returns a new list containing the results of applying `p` to the elements of `ls`. Do not use Racket's built-in `map` in your definition.

```
> (map add1 '(1 2 3 4))
(2 3 4 5)
```

8. Define and test a procedure `append` that takes two lists, `ls1` and `ls2`, and appends `ls1` to `ls2`.

```
> (append '(a b c) '(1 2 3))
(a b c 1 2 3)
```

9. Define and test a procedure `reverse` that takes a list and returns the reverse of that list.

```
> (reverse '(a 3 x))
(x 3 a)
```

10. Define and test a procedure `fact` that takes a natural number and computes the factorial of that number. The factorial of a number is computed by multiplying it by the factorial of its predecessor. The factorial of 0 is defined to be 1.

```
> (fact 0)
1
> (fact 5)
120
```

11. Define and test a procedure `memv` that takes an element and a list and returns the first sublist whose `car` is `eqv?` to the element, or `#f` if the element is absent from the list.

```
> (memv 'a '(a b c))
(a b c)
> (memv 'b '(a ? c))
#f
> (memv 'b '(a b c b))
(b c b)
```

12. Define and test a procedure `fib` that takes a natural number `n` as input and computes the *n*th number, starting from zero, in the Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, ...). Each number in the sequence is computed by adding the two previous numbers.

```
> (fib 0)
0
> (fib 1)
1
> (fib 7)
13
```

13. The expressions `(a b)` and `(a . (b . ()))` are equivalent. Using this knowledge, rewrite the expression `((w x) y (z))` using as many dots as possible. Be sure to test your solution using Racket's `equal?` predicate. (You do not have to define a `rewrite` procedure; just rewrite the given expression by hand and place it in a comment.)

14. Define and test a procedure `binary->natural` that takes a flat list of `0`s and `1`s representing an unsigned binary number in reverse bit order and returns that number. For example:

```
> (binary->natural '())
0
> (binary->natural '(0 0 1))
4
> (binary->natural '(0 0 1 1))
12
> (binary->natural '(1 1 1 1))
15
> (binary->natural '(1 0 1 0 1))
21
> (binary->natural '(1 1 1 1 1 1 1 1 1 1 1 1 1))
8191
```

15. Define subtraction using natural recursion. Your subtraction function, `minus`, need only take nonnegative inputs where the result will be nonnegative.

```
> (minus 5 3)
2
> (minus 100 50)
50
```

16. Define division using natural recursion. Your division function, `div`, need only work when the second number evenly divides the first. Division by zero is of course bad data.

```
> (div 25 5)
5
> (div 36 6)
6
```

17. Define a function `append-map` that, similar to `map`, takes both a procedure `p` of one argument a list of inputs `ls` and applies `p` to each of the elements of `ls`. Here, though, we mandate that the result of `p` on each element of `ls` is a list, and we `append` together the intermediate results. Do not use Racket's built-in `append-map` in your definition.

```
> (append-map countdown (countdown 5))
(5 4 3 2 1 0 4 3 2 1 0 3 2 1 0 2 1 0 1 0 0)
```

18. Define a function `set-difference` that takes two flat sets (lists with no duplicate elements) `s1` and `s2` and returns a list containing all the elements in `s1` that are **not** in `s2`.

```
> (set-difference '(1 2 3 4 5) '(2 4 6 8))
(1 3 5)
```

Brainteasers

19. In mathematics, the power set of any set S, denoted P(S), is the set of all subsets of S, including the empty set and S itself.

$$S = \{x, y, z\}$$

$$\mathcal{P}(S) = \{\{\}, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}.$$

The procedure `powerset` takes a list and returns the power set of the elements in the list. The exact order of your lists may differ; this is acceptable.

```
> (powerset '(3 2 1))
((3 2 1) (3 2) (3 1) (3) (2 1) (2) (1) ())
> (powerset '())
(())
```

20. The *cartesian-product* is defined over a list of sets (again simply lists that by our agreed upon convention don't have duplicates). The result is a list of tuples (i.e. lists). Each tuple has in the first position an element of the first set, in the second position an element of the second set, etc. The output list should contains all such combinations. The exact order of your tuples may differ; this is acceptable.

```
> (cartesian-product '((5 4) (3 2 1)))
((5 3) (5 2) (5 1) (4 3) (4 2) (4 1))
```

21. Rewrite some of the natural-recursive programs from above instead using `foldr` [http://docs.racket-lang.org/reference/pairs.html?q=foldr#%28def._%28%28lib._racket%2Fprivate%2Flist..rkt%29._foldr%29%29]. That is, the bodies of your definitions should not refer to themselves. The names should be the following:

- `insertR-fr`
- `filter-fr`
- `map-fr`
- `append-fr`
- `reverse-fr`
- `binary->natural-fr`
- `append-map-fr`
- `set-difference-fr`
- `powerset-fr`
- `cartesian-product-fr`

If you would like, I can recommend a treatise on fold operators containing answers to several of the above sub-problems [http://eprints.nottingham.ac.uk/224/1/fold.pdf]. It will also teach you about programming with `foldr`. There are *stunningly* beautiful definitions of the last two sub-problems. They're just mind-blowing. And to tease you further, know that some (pretty clever, albeit) folk solved this almost **50** years ago, back when lexical scope wasn't a thing, higher-order functions weren't commonplace like they are today, and many of the common programming idioms and that we take for granted just weren't around. Since I hate to pass up an excuse to show off something cool, I gotta tell you about the derivation and explanation of the last couple of answers here [http://ojs.statsbiblioteket.dk/index.php/brics/article/download/21934/19359], but you have to promise (1) you'll try it first on your own, and (2) that if you peek at the answers, you'll read the whole thing. It's short, moves quickly, and very high enlightenment/text ratio. That's my sales pitch.

22. Consider a function `f` defined as below

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

It is an open question in mathematics, known as the Collatz Conjecture [http://en.wikipedia.org/wiki/Collatz_conjecture], as to whether, for every positive integer `n`, `(f n)` is 1.

Your task is to, complete the below definition of `collatz`. `collatz` should be a function which will, when given a positive integer as an input, operate in a manner similar to the mathematical description above.

```
(define collatz
  (letrec
    ((odd-case
       (lambda (recur)
         (lambda (x)
           (cond
            ((and (positive? x) (odd? x)) (collatz (add1 (* x 3))))
            (else (recur x))))))
     (even-case
       (lambda (recur)
         (lambda (x)
           (cond
            ((and (positive? x) (even? x)) (collatz (/ x 2)))
            (else (recur x))))))
     (one-case
       (lambda (recur)
         (lambda (x)
           (cond
            ((zero? (sub1 x)) 1)
            (else (recur x))))))
     (base
       (lambda (x)
         (error 'error "Invalid value ~s~n" x))))
    ... ;; this should be a single line, without lambda
    ))
```

Your completed answer should be very short. It should be no more than one (prettily-indented) line long, and should not use lambda. Your `collatz` should compute the collatz of positive integers; for non-positive integers, it should signal an error “Invalid value”.

```
> (collatz 12)
1
> (collatz 120)
1
> (collatz 9999)
1
```

Just Dessert

21. A `quine` is a program whose output is the listings (i.e. source code) of the original program. In Racket, `5` and `#t` are both quines.

```
> 5
5
```

```
> #t
#t
```

We will call a quine in Racket that is neither a number nor a boolean an *interesting Racket quine*. Below is an interesting Racket quine.

```
> ((lambda (x) (list x (list 'quote x)))
  '(lambda (x) (list x (list 'quote x))))
((lambda (x) (list x (list 'quote x)))
  '(lambda (x) (list x (list 'quote x))))
```

Write your own interesting Racket quine, and define it as `quine`. The following should then be true.

```
> (equal? quine (eval quine))
#t
> (equal? quine (eval (eval quine)))
#t
```

Not every Racket list is a quine; Racket's standard printing convention will prepend a quote to a list. Make sure to use the above tests.