Assignment 1: Recursion and Higher-Order Functional Abstractions

> Recursion is the root of computation since it trades description for time.
>
> —*Alan Perlis*

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.
- To get our same answers, you'll want to either set `(print-as-expression #f)` in your file, or in DrRacket go to Language>Output Language>Output Syntax and change the Output Style radio button from `print` to `write`.
- **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 C311.zip archive and a1-student-tests.rkt test file. To use these tools, do the following:

```
1. The raco program comes bundled with your racket installation, add it to your system path.
2. Download and unzip the C311 course archive to some permanent location of your choosing.
3. In that location run ''raco link </path/to/C311/>''
4. Download a1-student-tests.rkt to the directory where you are writing your assignment a1.rkt.
5. Then, from within racket you should be able to
```

```
> (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 Oncourse. 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.

```
> (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 `occurs-?s` that takes a list and returns the number of times the symbol `?` occurs in the list.

```
> (occurs-?s '(? y z ? ?))
3
```

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 of equal length and forms a new list, each element of which is a pair formed by combining the corresponding elements of the two input lists.

```
> (zip '(1 2 3) '(a b c))
((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 every natural number less than it.

```
> (fact 5)
120
```

11. Define and test a procedure `member-?*` that takes a (potentially deep) list and returns `#t` if the list contains the symbol `?`, and `#f` otherwise.

```
> (member-?* '(a b c))
#f
> (member-?* '(a ? c))
#t
> (member-?* '((a ((?)) ((c) b c))))
#t
```

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. (The "direct" solution to this problem is very inefficient; see the second brainteaser for a more efficient version.)

```
> (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
```

Brainteasers

17. Rewrite some of the natural-recursive programs from above instead using `foldr`. That is, the bodies of your definitions should not refer to themselves. The names should be the following:

- `insertR-fr`
- `occurs-?s-fr`
- `filter-fr`
- `zip-fr`
- `map-fr`
- `append-fr`
- `reverse-fr`
- `binary->natural-fr`

18. Write another variant of the `fact` procedure, `fact-acc`, that is properly tail-recursive. That is, any last operation performed by the function is a recursive call (the tail call), or returns a value without recursion. (Hint: `fact-acc` must take two arguments.)

19. The following recursive algorithm computes x^n for a non-negative integer `n`:

$$\text{Power}(x, n) = \begin{cases} 1, & \text{if } n = 0 \\ x \times \text{Power}(x, n-1), & \text{if } n \text{ is odd} \\ \text{Power}(x, n/2)^2, & \text{if } n \text{ is even} \end{cases}$$

Write a Racket procedure `power` that uses this algorithm to raise a base `x` to a power `n`. For example:

```
> (power 2 0)
1
> (power 2 2)
4
> (power 2 10)
1024
> (power 10 5)
100000
> (power 3 31)
617673396283947
> (power 3 32)
1853020188851841
```

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

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

```
> (natural->binary 21)
(1 0 1 0 1)
> (natural->binary 8191)
(1 1 1 1 1 1 1 1 1 1 1 1 1)
```

You can solve this problem multiple ways, but you may want to look up quotient/remainder in the Racket documentation.

21. 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, as to whether, for every positive integer `n`, `(f n)` is 1.

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

```
(define base
  (lambda (x)
    (error 'error "Invalid value ~s~n" x)))

(define odd-case
  (lambda (recur)
    (lambda (x)
      (cond
        ((odd? x) (collatz (add1 (* x 3))))
        (else (recur x))))))

(define even-case
  (lambda (recur)
    (lambda (x)
      (cond
        ((even? x) (collatz (/ x 2)))
        (else (recur x))))))

(define one-case
 (lambda (recur)
   (lambda (x)
     (cond
       ((zero? (sub1 x)) 1)
       (else (recur x))))))
```

Your solution should use all of the provided functions, and should be no more than a single line long.

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

Just Dessert

22. 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`.