Assignment 6
Due: Sunday, May 17, 11:59PM

1. **10 points.** 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. **10 points.** 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.

3. **10 points.** 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. 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. Sample solution: 5 lines

4. **10 points.** Write a stream `funny-number-stream` that is like the stream of natural numbers (i.e., 1, 2, 3, ...) except numbers divisible by 6 are negated (i.e., 1, 2, 3, 4, 5, -6, 7, 8, 9, 10, 11, -12, 13, ...).

5. **15 points.** 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.

6. **15 points.** 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.

7. **15 points.** 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.

8. **15 points.** We now turn to the Racket's "quotation" mechanism. Consider the following example:

   ```
   (quote (+ (+ 2 3) 1))
   ```

   We expect the `cadr` of this expression to be `(+ 2 3)`. Now suppose we would like to return the list `(+ 5 1)`. That is, we want to evaluate only the `cadr` of our expression. Racket provides us a mechanism, called "quasiquote", which allows us to do just that: [1]

   ```
   (quasiquote (+ (unquote (+ 2 3)) 1))
   ```

   That is, as intended, this will return the expression

   ```
   `(+ 5 1).
   ```

   Write a translator function called TR, with one argument E, such that

   ```
   (TR E)
   ```

   produces a new expression E' with every occurrence of the symbol + in E replaced by the symbol *, and every occurrence of the symbol * in E is replaced with the symbol +.

   For example,

   ```
   (TR '(+ 1 2))
   ```

   returns the expression:

[1] Again, Racket provides concise notation for quasiquotes: `(+ ,(+ 2 3) 1), corresponds to (quasiquote (+ (unquote (+ 2 3)) 1)). Notice that this is not the 'normal' quotation mark but a backtick (usually located above the tab key on your keyboard.) instead. Again, for the rest of the assignment, the two notations will be used interchangeably

```
'(* 1 2)
```

and

```
(TR '(+ (* 2 3) (* (+ 1 2) (* 8 8))))
```

produces

```
'(* (+ 2 3) (+ (* 1 2) (+ 8 8)))
```

Use Racket's quasiquote mechanism in returning the result.

You can run the result of your translator using the eval function:

```
> (eval (TR '(+ (* 2 3) (* (+ 1 2) (* 8 8)))))
90
```

You may use the following skeleton:

```
(define (TR e)
  (cond
    [(number? e) .........]
    [(equal? (car e) '+)   ...........................]
    [(equal? (car e) '*)  `(+ ,(TR (cadr e))........................]
    [#t  (error "Error: Not a valid E term")]))
```