CSE 341, Spring 2018, Assignment 1
Racket Warmup
Due: Monday April 2, 10:00pm

20 points total (2 points each for Questions 1–3, 4 points for Question 4, 10 points for Question 5); up to 10% extra for the extra credit problem.

Include appropriate unit tests for each of your top-level functions.

You can use up to 2 late days for this assignment.

1. Write a *recursive* function `squares` that takes a list of numbers, and returns a new list of the squares of those numbers.

2. Write another version of the `squares` function, called `map-squares`, that uses the built-in `map` function in Racket. `map-squares` itself should not be recursive. Don't define a named helper function to compute each square — use an anonymous function.

3. Write a `ascending` function to test whether a list of integers is in strict ascending order. For example, `(ascending '(1 2 5))` should return `#t`, while `(ascending '(2 4 1))` and `(ascending '(2 2))` should both return `#f`. You should handle the empty list, and a list of one number. (These are both considered strictly ascending.) Also say in a comment whether or not your `ascending` function is tail-recursive.

4. A `let*` expression in Racket can be replaced by an equivalent set of nested `let` expressions. For example:

   ```
   (let* ([x 3]
          [y (+ x 1)]
          [z (+ x y)])
     (+ x y z))
   ```

   can be replaced by the equivalent expression:

   ```
   (let ([x 3])
     (let ([y (+ x 1)])
       (let ([z (+ x y)])
         (+ x y z))))
   ```

   Write a function that takes a list representing a `let*` expression and returns a new list representing the equivalent expression using `let`. (If you like showing that Racket is tolerant of strange characters in symbols, you could call your function `let*->let`, but if this is too peculiar for your liking you could call it `star-remover` or something else.) For example,

   ```
   (let*->let '(let* ([x 3] [y 4]) (+ x y)))
   ```

   should return

   ```
   '(let ((x 3)) (let ((y 4)) (+ x y)))
   ```

(Note that your output will just be printed using ordinary parentheses, with no square brackets.)

Hints: it's legal to have no variables bound in a `let*`, so for example your function should handle `(let* () (+ x 1))`. You can turn this into `(let () (+ x 1))`. It's also legal to have more than one expression in the body of the `let*`. (Having more than one expression in the body is only useful if you have side effects, but your function should handle it.)

Include some unit tests that check that your function returns exactly the right list structure.

You can also check that the resulting `let` is syntactically correct and compute the correct value by pasting it into the Racket interaction pane and evaluating it. If you want to add a unit test or two for this, you can do so like this:

```
(eval '(let* ([x 3] [y (+ x 1)]) (+ x y)) (make-base-namespace))

(eval (let*->let '(let* ([x 3] [y (+ x 1)]) (+ x y))) (make-base-namespace))
```

Both of these should evaluate to 7. (The "namespace" argument is needed if you are writing these expressions in definitions rather than in the interaction pane.)

5. This question is based on the symbolic differentiation program linked from the 341 Racket web page. That program adapts an example in Chapter 2 of the book *Structure and Interpretation of Computer Programs*. The full text is available online (linked from the Racket page), although just the program and class discussion should be enough for this assignment. The symbolic differentiation program linked from the 341 web page is somewhat modified however: it doesn't use constructor functions, and it separates out finding the derivative from simplifying expressions.

   First, load the symbolic differentiation program into Racket and try it, to make sure it is working OK and that you understand it. Now add the following extensions, by allowing the expressions to be differentiated to include:

   - the minus operator
   - sin and cos
   - raising an expression to an integer power

   Use the names of the built-in Racket functions for each of these: `- sin cos expt`. You can assume that minus always has two arguments.

   Minus should be really easy — use sum as a model. The rules for the others are as follows:

$$\frac{d(\sin u)}{dx} = \cos u(\frac{du}{dx})$$

$$\frac{d(\cos u)}{dx} = -\sin u(\frac{du}{dx})$$

$$\frac{d(u^n)}{dx} = nu^{n-1}(\frac{du}{dx})$$

   For sin and cos, just simplify applying these to a number, e.g., $\sin 0$ should simplify to 0. For simplification of the power function, build in the rules that anything to the 0 power is 1 (including 0^0), and anything to the power 1 is itself. Also simplify a number raised to another number, e.g., 3^2 should simplify to 9. Your code for this question should be written entirely in a functional style — no side effects.

Extra Credit: Write a function that converts an arbitrary Racket expression, which might contain zero or more `let*` expressions, into an equivalent expression using `let`. To simplify the problem a little, you can assume that the symbol `let*` is only used in a `let*` expression. For example, your function should work with expressions like this:

```
(define (squid z)
  (let* ([x 3]
         [y x])
    (+ x y z)))
```

and also:

```
(define (ugly z)
  (let* ([x (let* ((y (* 2 z))) (* 2 y))]
         [y x])
    (let* ([x x])
      (+ x y z))))
```

(I hope you never write anything as ugly as `ugly`! The point is that the `let*` can occur in various places.)

Turnin: Your program should include some well-chosen unit tests for each of your functions. Your program should be tastefully commented (i.e. put in a comment before each function definition saying what it does, but don't overcomment), and in good style. You can use either one file that includes both the functions and unit tests, or two files, one with the functions and the other with your tests.