The purpose of this assignment is to give you experience with writing a larger program in Racket, and also with using Racket macros. You can (and should) use side effects, with care, in Question 3. Don’t use side effects for the answers to any of the other questions.

Points: 15 points Question 1, 4 points each Questions 2, 3 and 4.

You can use up to 3 late days for this assignment.

Turnin: Turn in two files: polynomials.rkt, with the functions and unit tests for Questions 1 and 2; and macros.rkt, with the functions, macros, and unit tests for Questions 3 and 4. You don’t need to turn in sample output — the unit tests are enough for those. As before, your program should be tastefully commented (i.e. put in a comment before each function definition saying what it does). Style counts!

1. Write and test a Racket function `poly-multiply` that multiplies two polynomials in a symbolic variable and returns the result. The polynomials should be represented as lists of terms, where each term is in turn a list consisting of a coefficient and an exponent of the symbolic variable.

The polynomials should be normalized: they should be sorted with the largest exponent first, there shouldn’t be two terms with the same exponent, and that shouldn’t be any terms with a coefficient of 0. You can assume that the exponents will be non-negative integers. The zero polynomial is represented as the empty list. The result returned by `poly-multiply` should be normalized as well.

For example:

```
(poly-multiply '((1 3) (1 2) (1 1) (1 0)) '((1 1) (-1 0)))
```

should evaluate to `((1 4) (-1 0))`.

In standard algebraic notation, this represents

\[ x^4 - 1 = (x^3 + x^2 + x + 1) \cdot (x - 1) \]

Here are some other polynomial pairs that you can turn into unit tests for your function:

\[-3x^4 + x + 5 \cdot 0\]
\[0 \cdot x^2\]
\[(x^3 + x - 1) \cdot -5\]
\[(-10x^2 + 100x + 5) \cdot (x^{999} - x^7 + x + 3)\]
\[3 \cdot x\]
\[x \cdot 3\]

2. Define a function `poly->code` that converts a polynomial in normalized form as in Question 1 into evaluable Racket code. It should take two parameters: the list representing the polynomial, and the symbolic variable. Here are some examples:

```
(poly->code '((1 3) (5 2) (7 1) (10 0)) 'x) =>
'(+ (expt x 3) (* 5 (expt x 2)) (* 7 x) 10)
```

```
(poly->code '((1 1) (-10 0) 'x) =>
'(+ x -10)
```
(poly->code '((1 1)) 'x) => 'x
(poly->code '((10 0)) 'x) => 10
(poly->code '((1 0)) 'x) => 1
(poly->code '() 'x) => 0

So in general, the result will be a sum of terms, but if there is just one term, just return that term. In
general a term is the coefficient times the variable raised to the exponent; but omit the coefficient if it
is 1 and the exponent isn’t 0, omit the exponent if it is 1, and omit both the variable and the exponent
if the exponent is 0. Finally, return 0 for the zero polynomial.

Add appropriate unit tests for your poly->code function.

Since the result returned from poly->code is legal Racket code, we should be able to evaluate it.
Suppose we define two polynomials p1 and p2 and a number x in the interaction pane:

(define p1 '((1 3) (1 2) (1 1) (1 0)))
(define p2 '((1 1) (-1 0)))
(define x 4)

Then both of these expressions should evaluate to the same number, namely 255:

(eval (poly->code (poly-multiply p1 p2) 'x))
(* (eval (poly->code p1 'x)) (eval (poly->code p2 'x)))

(If you use these examples in the definition pane, for example in unit tests, you’ll need to use an
additional namespace argument to eval.)

3. Racket macros: the lecture notes and code for delay and force include functions my-delay and
my-force. Rewrite my-delay as a macro, so that the user doesn’t have to manually wrap the delayed
expression in a lambda. So the syntax for my-delay should be just like Racket’s delay. Note in
particular that you can have multiple expressions in the body. For example, this should work:

(my-delay (write "hi there ") (+ 3 4))

Rewrite the my-force function from the lecture notes if necessary so that it works correctly with your
my-delay macro. (Or perhaps it will be OK as is.) Leave my-force as a function in any case, rather
than making it a macro.

For the unit tests for my-delay and my-force, you want to demonstrate that the expressions aren’t
evaluated when you construct the delay; that they are evaluated the first time you use my-force and
that it returns the correct value; and that additional uses of my-force continue to return the correct
value without re-evaluating the expressions. You can use side effects judiciously here. The tests need
to be automated — do things like checking the value of a counter, rather than expecting the user to
look at the output and see if something got printed.

4. Another Racket macro: define a macro my-and that does exactly the same thing as the built-in Racket
and. (Hint: see the handouts for macros, in particular the my-or example. Remember that and works
on an indefinite number of expressions, including 0 expressions.) Include suitable unit tests.