INTRODUCTION TO SYMBOLIC DIFFERENTIATOR—EXPRESSIONS

We are going to build a system for processing basic algebraic expressions, including differentiation.

Let’s get started by with some decisions about representation:

• Numbers will simply be Scheme numbers—the underlying numeric object.
• Variables will be Scheme symbols—e.g., 'x will be the representation for the variable x.

Both a single number (like 3) and a single variable (like 'x) are valid expressions in our system.

Question 1. Let’s write some predicates for numbers.

We can use Scheme’s built-in number? because we’re using Scheme-numbers to represent numbers. But let’s also write =number?, which an expression (first param) and a number (second param), and and tests if the expression is a number and if it’s equal to the second param. E.g.:

(=number? 3 3) → #t
(=number? 3 4) → #f
(=number? 'x 3) → #f

Write =number?. Make sure to remember to test whether expr is a number—don’t assume that it is!

(define (=number? expr n))

Question 2. Let’s write some predicates for variables.

To test if an expression is a variable, all we have to do is ask it if is a symbol; i.e.:

(define (variable? expr)
  (symbol? expr))

Now, write an predicate to test if two expressions are the same variable. E.g.:

(same-variable? 'x 'x) → #t
(same-variable 'x y) → #f
(same-variable 'x 3) → #f
Write the procedure same-variable, remembering that its inputs might not be variables at all. Make sure to use the variable? predicate in your implementation.

(define (same-variable expr1 expr2) )

**Compound expressions**, for now, will be prefix with the symbol '+ for plus, '* for multiplication, and '** for exponentiation. Also, initially, expressions must be binary, so '(* 3 x) is a valid expression (for 3x) and '(+ 3) is not.

Here is an initial set of a constructor, predicate, and selectors for sum-expressions:

(define (make-sum expr1 expr2)  
  (list '+ expr1 expr2))

(define (sum? expr)  
  (and (pair? expr) (eq? (car expr) '+))) ; tests it's a list first!

(define (addend s) (cadr s)) ; must check it's a sum-expr prior to
(define (augend s) (caddr s)) ; using either of these

**Question 3.** Let’s improve the constructor so that it does simplifications when possible. E.g., if the two expressions are both numbers, it should add them and return the sum: (make-sum 3 5) should return 8, not '(+ 3 5). Also, adding zero to any expression should result in that same expression; e.g. (make-sum 0 'x) should be 'x, not '(+ 0 x). Write a better version of make-sum below:

(define (make-sum expr1 expr2) )