CS 441
Program 3
Expression evaluation, with environment

For the last programming assignment of the semester, we're going to apply what we've learned about implementation to build part of an interpreter. This will require you to maintain a state, defining what variables have been defined. (This can just be a list of name-value pairs, or a hash. You do not have to use lexical addressing.)

You are given a working implementation of a numeric expression evaluator, using Maybe. Recall that a Maybe returns one of two things: either `(just x)`, where x is the return value, or `#<nothing>`. This will not be enough for our needs. You should begin by modifying this code so that it uses an Either (sometimes called a Result), which returns either `(success x)` or `(failure x)`. The labels success and failure are predefined for you. The label is followed by 1 thing, but that 1 thing can be a list. There are functions `(from-success v x)` and `(from-failure v x)`, corresponding to the from-just function. Like from-just, they require a dummy value to return if the actual data is missing—something we hope to avoid.

Once you have that done, you're ready to begin the main part of the program. A programming language that doesn't allow variables is very limited—in fact, it's nothing but a calculator with cumbersome notation. Your task is to include a state, consisting of variables that have been declared, and their values. This will be passed into the evaluator, and our expressions can now include variables.

Your finished problem is a simple REPL loop: Present a prompt for the user to enter an expression. Evaluate each expression, presenting the result (success with the value, or failure with an error message, and the current state) for each. Continue until the user decides to quit.

Deliverables: As for the previous projects:
1) Working code (.rkt file(s))
2) LLM Prompts
3) Any machine-generated code that was included in your program, with or without modification
4) A short reflection document: which LLM did you use? What did you use it for? Where was it helpful, where was it not? How has using the LLMs this semester helped you program better? Do you feel like it has hindered you in some ways? What additional skills do you need to use these tools more effectively? As we near the end of the semester, you've had some experience using these tools; what's the big picture you're taking away from the experience? What advice do you have for future students, or for faculty devising courses?

Detailed specification & grammar:

As you can tell from inspecting the sample expressions with the provided code, expressions use a Racket-style notation, with numeric values tagged and the operations `add, sub, mult, div` defined. Recall that the quote is used to indicate data rather than a function call:

```
(num 5)  ; a call to function 'num' with parameter 5
'(num 5) ; a data item
```

We’re going to introduce a new type, an id. These can be declared, assigned, and removed. Your evaluator will take the state as one of its parameters, and return the updated state as part of its return value.

Definitions: An id must begin with an alphabetic character. The leading character is followed by 0 or more letters, digits, hyphens (-) or underscores (_), in any combination. The following operations are allowed:

`‘(define a)` ; this creates a new variable a, giving it the value `undefined`. Any use of an undefined value in an expression causes a failure. If this ID is already defined, a failure results.
`‘(define a expr)` ; this creates a new variable a, giving it the value 5. If the expression fails to evaluate, or the ID is already in use, a failure results. Example:
`‘(define b (add (id a) (num 1)))` ; this creates a new value b, giving it the value 1 more than a has. Note that it follows the same syntax used in other expressions.

Note the pattern: for an expression `expr`, `(first expr)` will be the operation; for a `define`, `(second expr)` will be the variable name, and `(third expr)` the value to be assigned.

```
(assign a expr) ; evaluates expr and assigns its value to a.
;    a must already be defined. (if it isn’t, a failure results)
```

Note that
`(assign a (add (id a) (num 1)))`
is a valid assignment and will cause no problem; the expression will be evaluated with the current value of a, and the new value assigned afterwards.

Once these are done, we can build expressions such as
`‘(add (num 5) (id a))`

We can also carry out the operation
`(remove id)`
which removes the variable with that id from the current state. If that id does not correspond to a variable, a short message is printed but no other operation is carried out. (It does not cause a failure.)

```
(remove d)
Error: remove d: variable not defined, ignoring
```
(Note that “Error:” is just part of the error message; a Racket error [an exception] is not thrown.)

MANAGING STATE, FUNCTIONALLY

Your state will be the collection of defined id’s, and their values. You can store these in a list, a hash, or some other convenient data structure. Most of the standard data structures are already defined in Racket. Use the functional (immutable) version rather than the mutable version.

The basic idea is simple: In addition to the expression to be evaluated, the eval function will also get the current state. On a success, it will return the value it usually would, along with an updated version of the state (which may or may not have been changed). On a failure, it will return an appropriate message, along with the original state it was passed, unchanged. If the failure is passed back up the call chain, the function that receives the failure can just pass it along.