CSCI 301, Lab # 3

Fall 2024

Goal: This lab begins a series of labs that will build an interpreter for Scheme. In this lab we get used to recursion as an evaluation strategy, and programming it. Our programs will be represented by lists, to avoid parsing problems until later.

Submission: You will submit your program, named `eval.rkt`, to Canvas. Tests are provided, but no skeleton file. As in prior labs, please make sure your file has a block comment at the top with your name, etc., and each function has a comment describing its purpose, arguments, and return value.

Unit tests: At a minimum, your program must pass the unit tests found in the file `lab3-test.rkt`. There should be no output.

Evaluation: The process of evaluating a Scheme expression that consists only of function calls (procedure applications), for example `(+ 3 x (+ 2 2) (* 2 (+ 4 1)))`, is really very simple. The process follows three simple rules:

1. Numbers evaluate to themselves.

   To check whether something is a number in scheme, use the `number?` predicate.

2. Symbols, such as `cons`, `+`, and `x`, are looked up in a table called the *environment*. We will use a list for our environment.

   To check whether something is a symbol in scheme, use the `symbol?` predicate. Do you see a pattern here?

3. Lists are evaluated recursively. First, each element of the list is evaluated, and then the first argument, which should be a procedure at this point, is applied to the evaluated arguments:

```
(        +        3 x (+ 2 2) (* 2 (+ 4 1)))
         ⇓        ⇓ ⇓    ⇓          ⇓
(#<procedure:+> 3 5     4          10       )
         ⇓
        22
```

To implement this, you will write at least two procedures: `lookup` and `evaluate`.

Lookup: The procedure `lookup` finds the value of a variable in the environment. We represent an environment as a list of lists. Each sublist holds a variable, and the value of that variable. For example:

```
(define env (list (list ’x 5)
                  (list ’+ +)
                  (list ’* *)))
```

This would be enough of an environment to evaluate the above expression.

The `lookup` procedure takes two arguments: a symbol and an environment, and

- If the first argument is not a symbol, returns an error.
- If the symbol is not in the environment, returns an error.
- Otherwise, returns the value bound to the variable.

`lookup` should be written as a simple recursion through the environment, comparing the provided symbol with the symbol in the `car` (first element) of each variable-value sublist. If it finds a matching symbol, the `cadr` (the second element) is returned.

Evaluate: The `evaluate` procedure takes two arguments, an `expression` to evaluate, and an `environment`. It follows the above rules:

- A number is returned unchanged.
- For a symbol the return value is looked up in the environment.
- For a list, each element in the list is evaluated recursively by the `evaluate` procedure. You may want to use `map`, but it is not required.
  - If the first thing in the evaluated list is not a procedure, an error is returned. How do you think you check for a procedure in Scheme? Don't overthink it.
  - Otherwise, the procedure is applied to the evaluated arguments.
- If the expression is anything else, an error is returned.