CSCI 301, Lab # 4

Fall 2024

Goal: This is the second in a series of labs that will build an interpreter for Scheme. In this lab we process some special forms.

Submission: This lab builds on Lab 3. You will extend the functionality your Lab 3 program and submit `eval.rkt`. As usual, unit tests are provided. 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 `lab4-test.rkt`. Place this file in the same folder as your program, and run it; there should be no output.

Special forms: Special forms are signalled by certain keywords. Ultimately we will process the special forms `if`, `cond`, `let`, `lambda`, and `letrec`. For this lab, however, we will only do the two conditionals, `if` and `cond`.

You will add another case to your `evaluate` function that will check for special forms. So your `evaluate` function, adding in what you did last week, will be able to handle numbers, symbols, special forms, and lists that are not special forms. (It will help if you check for them in that order.)

For this week, a special form is a list where the `car` of the list is one of the two symbols, `if` or `cond`. Write a procedure to determine if an expression is a special form. Examples:

```
> (special-form? '(if 1 2 3))
#t
> (special-form? '(cond 1 2 3))
#t
> (special-form? '(+ 1 2 3))
#f
> (special-form? 99)
#f
> (special-form? 'if)
#f
```

Note that `special-form?` only checks to see if its argument is a list, and if the `car` of the list is `if` or `cond`. It doesn't check any more of the syntax of the form (but it could).

If your evaluation function finds a special form, it hands the whole form off to the `evaluate-special-form` function, which acts like this:

```
> (evaluate-special-form
    '(if (= 1 2) 3 4) e2)
4
```

The `evaluate-special-form` function checks to see if the `car` of the list is either `if` or `cond` (otherwise it should raise an error), and then does the appropriate thing.

If: The special form `if` is clearly not a function, because functions evaluate *all* their arguments. So, for instance, this:

```
> (if (= 1 1)
        (print "Hello")
        (print "Goodbye"))
"Hello"
```

would print *both* `Hello` *and* `Goodbye` if `if` had been defined as a function. Clearly, the `if` special form evaluates only its *first argument*, and, on the basis of whether or not that argument evaluates to `#f` (any value other than `#f` acts as *true*), evaluates *either* the second argument *or* the third, not both, accordingly. The provided environment is used to evaluate the subform. The one it chooses to evaluate is the value of the special form.

Cond: Once you've figured out how to do `if`, you can tackle `cond`. Consider a typical `cond` special form:

```
> (cond ((= 1 2) 3)
        ((= 4 5) 6)
        ((= 7 7) 8)
        (else 9))
8
```

The `cdr` of this list is clearly a list of alternatives. You will write a recursive procedure to go through this list, and evaluate each of the tests in the items. For example, the test in the item `((= 4 5) 6)` is `(= 4 5)`. As soon as a test evaluates to `#t`, the other item in the list is evaluated, and it is returned as the value of the special form. Again, use the provided environment to evaluate the selected forms.

But what about `else`? If we treat `else` as a condition that always evaluates to true, the `else` clause will behave as expected. We can accomplish this by simply putting the symbol `else` into the environment with value `#t`.