carlae

This lab is part 1 of a two week lab, which will be completed next week. Due dates for this part are:

- *Submission to website:* Monday, April 24, 10pm
- *Checkoff by LA/TA*: Thursday, April 27, 10pm

This lab assumes you have Python 3.5 or newer installed on your machine. It is alright to use modules that are already imported for you in the template, but you should not add any imports to the code skeleton.

Introduction

In this lab, you will implement an interpreter for a dialect of LISP, one of the earliest high-level programming languages (it was invented by John McCarthy at MIT in 1958!). Because this language will in some ways be quite small compared to Python, we'll call it *carlae*, after *Leptotyphlops carlae*. Its syntax is simpler than Python's, and the complete interpreter will fit in a single Python file. However, despite its small size, the language we will implement here will be *Turing Complete*, i.e., in theory, it will be able to solve any possible computational problem (so it would be possible, for example, to implement the *HyperMines* program from week 4 in *carlae*).

LISP and *carlae*

As with most LISP dialects, the syntax of *carlae* is far simpler than that of Python. All-in-all, we can define the syntax of *carlae* as follows:

- *carlae* programs consist solely of expressions. There is no statement/expression distinction.
- Numbers (e.g. `1`) and symbols (e.g. `x`) are called *atomic expressions*; they cannot be broken into pieces. These are similar to their Python counterparts, except that in *carlae*, operators such as `+` and `>` are symbols, too, and are treated the same way as `x` and `fib`.
- Everything else is an *S-expression*: an opening round bracket `(`, followed by zero or more expressions, followed by a closing round bracket `)`. The first subexpression determines what it means:
    - An S-expression starting with a keyword, e.g. `(if ...)`, is a *special form*; the meaning depends on the keyword.
    - An S-expression starting with a non-keyword, e.g. `(fn ...)`, is a function call, where the first element in the expression is the function to be called, and the remaining subexpressions represent the arguments to that function.

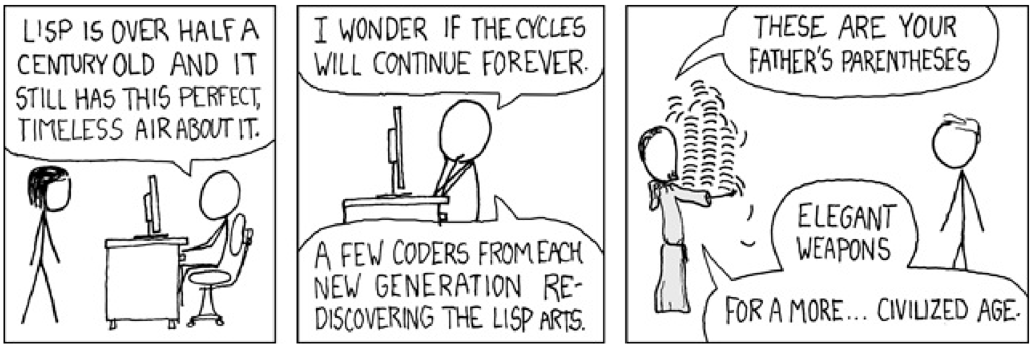
And that's it! The whole syntax is described by the three bullet points above. For example, consider the following definiion of a function that computes Fibonacci numbers in Python:

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

We could write an equivalent program in *carlae*:

```
(define fib
  (lambda (n)
    (if (<= n 1)
      n
      (+ (fib (- n 1)) (fib (- n 2)))
    )
  )
)
```

Using so many parentheses might take some getting used to, but it helps to keep the language simple and consistent. Some people have joked that LISP stands for "Lots of Insipid and Silly Parentheses," though some might argue instead that it stands for "Lisp Is Syntactically Pure" :)


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

Interpreter Design

Despite its small size, the interpreter for *carlae* will still be rather complicated. To help manage this complexity, we'll start with a very small language, and we'll gradually add functionality until we have a fully-featured language. As with most interpreters, we will think of our *carlae* interpreter as consisting of three parts:

- A *lexer*, which takes a string as input and produces a list of *tokens*, which represent meaningful units in the programming language.
- A *parser*, which takes the output of the lexer as input and produces a structured representation of the program as its output.
- An *evaluator*, which takes the output of the parser as input, and actually handles running the program.

Lexer (Tokenizer)

Our first job is *lexing* (or *tokenizing*). In *carlae*, we'll have exactly three kinds of tokens: opening round brackets `(`, closing round brackets `)`, and everything else (separated by whitespace). Your first task for the lab is to write a function called `tokenize`, which takes a single string representing a program as its input and outputs a list of tokens. For example, calling `tokenize("(foo (bar 3.14))")` should give us the following result: `['(', 'foo', '(', 'bar', '3.14', ')', ')']`

Unlike in Python, whitespace does not matter, so, for example, the `tokenize` function should produce exactly the same output for both of the following programs:

```
(define circle-area
  (lambda (r)
    (* 3.14 (* r r))
  )
)
```

```
(define circle-area (lambda (r) (* 3.14 (* r r))))
```

Your `tokenize` function should also handle comments. Comments in *carlae* are prefixed with a semicolon (`;`), instead of the octothorpe (`#`) used by Python. If a line contains a semicolon, the `tokenize` function should not consider that semicolon or the characters that follow it on that line to be part of the input program.

Implement the `tokenize` function in `lab.py`. After doing so, your code should pass test 1 in `test.py`.

Parser

Our next job is *parsing* the list of tokens into an *abstract syntax tree*, a structured representation of the expression to be evaluated. Implement the `parse` function in `lab.py`. `parse` should take a single input (a list of tokens as produced by `tokenize`) and should output a representation of the expression, where:

- a number is represented as an instance of `int` or `float`
- a symbol is represented as a string
- an S-expression is represented as a list of its parsed subexpressions

For example, the program above that defined `circle-area` should parse as follows:

```
['define', 'circle-area', ['lambda', ['r'], ['*', 3.14, ['*', 'r', 'r']]]]
```

Note that the structure of the parser's output is such that a recursive solution is likely the "path of least resistance."

In the case that parentheses are mismatched in the input, the function should raise a `SyntaxError`.

After implementing both `tokenize` and `parse`, you should pass tests 1-3 in `test.py`

Evaluator Strategy

> *"How do you eat a big pizza? One little bite at at time..."*
> -Anonymous

Now that we have the program in a form that is (relatively) easy to work with, we can move on to implementing the *evaluator*, which will handle actually running the program. This part of the interpreter will get fairly complicated, so we will start small, and add in more pieces later. We will make several "false steps" along the way, where we'll need to make modifications to pieces we had implemented earlier.

Because of this, it will be important to save backups of your `lab.py` file after every major modification (or to use a version control system such as Mercurial or Git), so that if something goes wrong, you can revert to a working copy.

Evaluator 1: Calculator

We'll hold off on implementing variables, lists, conditionals, and functions for a little while; for now, we'll start by implementing a small calculator that can handle the `+` and `-` operations.

Note that we have provided a dictionary called `carlae_builtins` in `lab.py`, which maps the names `+` and `-` to functions. Each of these functions takes a list as an argument and returns the appropriate value.

Define a function `evaluate`, which takes as its sole input an expression of the same form as the parser's output. `evaluate` should return the value of the expression:

- If the expression is a symbol of a name in `carlae_builtins`, it should return the associated object.
- If the expression is a number, it should return that number.
- If the expression is a list (representing an S-expression), each of the elements in the list should be evaluated, and the result of evaluating the first element (a function) should be called with the remaining elements passed in. The overall result of evaluating such a function is the return value of that function call.
- If the expression is a symbol that is not in `carlae_builtins`, or is a list whose first element is not a function, it should raise an `EvaluationError` exception.

Implement the `evaluate` function in `lab.py` according to the rules above. After doing so, you code should pass tests 1-4 in `test.py`.

Testing: REPL

It is kind of a pain to have to type out all of the arguments to `evaluate` each time we call it. As such, we'll implement a REPL (a "**R**ead, **E**valuate, **P**rint **L**oop) for *carlae*. A REPL has a simple job: it continually prompts the user for input until they type `QUIT`. Until then, it:

- accepts input from the user,
- tokenizes and parses it,
- evaluates it, and
- prints the result

If an error occurs during any of these steps, an error message should be displayed and that expression may be ignored, but the program should not exit.

To implement the REPL, we can make use of Python's built-in `input` function. `input` takes an argument representing a prompt to be displayed to the user and returns the string that they type (it is returned when they hit enter).

The following shows one possible interaction with a REPL, with a particular prompt and output formatting (you are welcome to use whatever formatting you like!):

```
in> (+ 2 3)
  out> 5

in> (+ 2 (- 3 4))
  out> 1

in> (- 3.14 1.14 1)
  out> 1.0000000000000004

in> (+ 2 (- 3 4 5))
  out> -4

in> QUIT
```

Implement a REPL for *carlae* and use it to test your evaluator. **The REPL can/should be one of your main means of testing moving forward**; feel free to try things out using the REPL as you work through the remainder of the lab. The functionality of your REPL will not be tested automatically; rather, it will be tested during the checkoff. The REPL should only start when the lab is run directly, not when `lab.py` is imported from another script.

Additional Operations

Implement two new operations: `*` and `/`:

- `*` should take arbitrarily-many arguments and should return the product of all its arguments.
- `/` should take arbitrarily many arguments. It should return the result of successively dividing the first argument by the remaining arguments.

After implementing the evaluator and the `*` and `/` operations, try them out in the REPL, and then by using `test.py`. At this point, your code should pass tests 1-5 in `test.py`.

Evaluator 2: Variables

Next, we will implement our first special form: *variable definition* using the `define` keyword.

A variable definition has the following syntax: `(define NAME EXPR)`, where `NAME` is a symbol and `EXPR` is an arbitrary expression. When *carlae* evaluates a `define` expression, it should associate the name `NAME` with the value that results from evaluating `EXPR`. In addition, the `define` expression should evaluate to the result of evaluating `EXPR`.

The following transcript shows an example interaction using the `define` keyword:

```
in> (define pi 3.14)
  out> 3.14

in> (define radius 2)
  out> 2

in> (* pi radius radius)
  out> 12.56

in> QUIT
```

Note that `define` differs from the function calls we saw earlier. It is a *special form* that does not evaluate the symbol that follows it; it only evaluates the expression the follows the name. As such, we will have to treat it differently from a normal function call.

In addition, in order to think about how to implement `define`, we will need to talk about the notion of *environments*.

Environments

Admitting variable definition into our language means that we need to be, in some sense, more careful with the process by which expressions are evaluated. We will handle the complexity associated with variable definition by maintaining structures called *environments*. An environment consists of bindings from variable names to values, and possibly a parent environment, from which other bindings are inherited. One can look up a name in an environment, and one can bind names to values in an environment.

The environment is crucial to the evaluation process, because it determines the context in which an expression should be evaluated. Indeed, one could say that expressions in a programming language do not, in themselves, have any meaning. Rather, an expression acquires a meaning only with respect to some environment in which it is evaluated. Even the interpretation of an expression as straightforward as `(+ 1 1)` depends on an understanding that one is operating in a context in which + is the symbol for addition. Thus, in our model of evaluation we will always speak of evaluating an expression *with respect to some environment*.

To describe interactions with the interpreter, we will suppose that there is a "global" environment, consisting of bindings of the names of built-in functions and constants to their associated values. For example, the idea that `+` is the symbol for addition is captured by saying that the symbol `+` is bound in this global environment to the primitive addition procedure we defined above.

One necessary operation on environments is looking up the value to which a given name is bound. To do this, we can follow these steps:

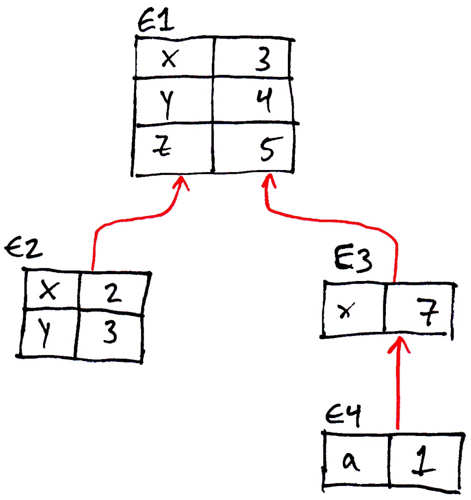
- If the name has a binding in the environment, that value is returned.
- If the name does not have a binding in the environment and the environment has a parent, we look up the name in the parent environment (following these same steps).
- If the name does not have a binding in the environment and the environment does not have a parent, an `EvaluationError` is raised.

Note that looking up a name in an environment is similar to looking up a key in a dictionary, except that if the key is not found, we continue looking in parent environments until we find the key or we run out of parents to look in.

In order to make variables work properly, you will need to implement the kind of lookup described above in Python. It is up to you to decide how to implement environments and the associated lookups; your implementation will not be tested directly by the automatic checker, but rather will be tested by looking at the end-to-end behavior of your evaluator. Regardless of how you implement environments, **you should make sure your environment representation can handle variables with arbitrary names**, and you should be prepared to discuss your implementation during the checkoff.

Environments: Example

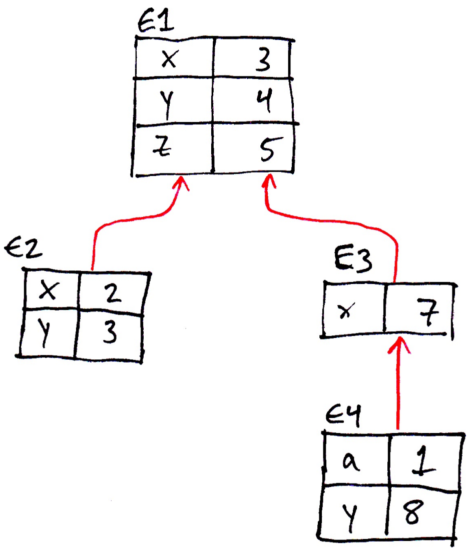
The following shows an example of an environment structure, where arrows indicate each environment's parent, if any. Here we have four environments, labeled **E1**, **E2**, **E3**, and **E4**. Both **E2** and **E3** have **E1** as a parent environment, and **E4** has **E3** as a parent environment. **E1** does not have a parent environment.



We say that the values `x`, `y`, and `z` are *bound* in **E1**. Note that `x` and `y` are bound in **E2**; `x` is bound in **E3**, and `a` is bound in **E4**. Looking up a name, we work our way up the arrows until we find the name we are looking for. For example, looking up `a` in **E4** gives us the value `1`, and looking up `z` in **E4** gives us `5`.

Notice that, as was mentioned above, the evnrionment is crucial for determining the result of an expression: evaluating `x` in **E1** gives us `3`; evaluating it in **E2** gives us `2`; and evaluating it is **E3** *or* **E4** gives us `7`.

If we were to evaluate `(define y 8)` in **E4**, this would result in a new binding inside of **E4** (without affecting the parent environments):

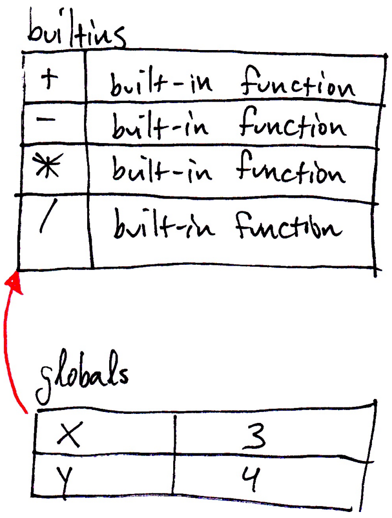


Environments: Initial Structure

For purposes of our REPL, we will start by thinking about two main environments: an environment to hold the *built-in* values (such as the `+` function), and a "global" environment where top-level definitions from users' programs will be bound. For example, running the code below from the REPL should result in the environment structure shown in the picture that follows:

```
in> (define x 3)
  out> 3

in> (define y 4)
  out> 4
```



Evaluator Changes

Implement a Python structure for representing an environment, and modify the `evaluate` function so that it handles variables and the `define` keyword. Beyond implementing a representation for environments, we will need to make four modifications to our `evaluate` function. We will need to:

- modify `evaluate` so that it takes a second (optional) argument: the environment in which the expression should be evaluated. If no environment is passed in, an empty environment should be used, whose parent is an environment containing all the bindings from the `carlae_builtins` dictionary.
- make sure that `evaluate` handles the `define` keyword properly, evaluating the given expression and storing the result in the environment that was passed to `evaluate`.
- modify the way symbols are handled in `evaluate`, so that if the symbol exists as a key in the environment (or a parent environment), `evaluate` returns the associated value.

Note that after implementing these changes, you can test your implementation using the examples above in the REPL.

To make automatic checking possible, define a function called `result_and_env` that takes the same arguments as `evaluate` but returns a tuple with two elements: the result of the evaluation, and the environment in which the expression was evaluated. **Your code will not pass the tests without this function.**

After implementing variable definition and lookup (and `result_and_env`), try it out in the REPL, and then in `test.py` (at this point, your code should pass tests 1-8 in `test.py`).

Evaluator 3: Functions

So far, we have a pretty nice calculator, but there are a few things missing before we can really call it a programming language. One of those things is *user-defined functions*.

Defining functions

Currently, the operations we can perform are limited to the functions in the `carlae_builtins` dictionary. We can really empower a user of the language by allowing them to define functions of their own. We will accomplish this via the `lambda` special form (so called because it is strongly rooted in Church's lambda calculus).

A `lambda` expression takes the following form: `(lambda (PARAM1 PARAM2 ...) EXPR)`. The result of evaluating such an expression should be an object representing that function (note that this statement represents a function *definition*, not a function *call*). Importantly, there are a few things we need to keep track of with regard to functions. We need to store:

- the code representing the body of the function (which, for now, is restricted to a single expression representing the return value)
- the names of the function's parameters
- a pointer to the environment in which the function was defined

Once again, it is up to you to determine how exactly a function should be represented in your interpreter; but it is important that, however it is represented, it stores the information above (and also that you are able to distinguish it from the other syntactic forms we have seen so far).

For example, the result of evaluating `(lambda (x y) (+ x y))` in the global environment should be an object that stores the following information:

- the function's parameters, in order, are called `x` and `y`.
- the function's body is the expression `(+ x y)`.
- the function was defined in the global environment.

Calling Functions

We also need a way to *call* user-defined functions. When the first element in an S-expression evaluates to a user-defined function, we will need to call the function by taking the following steps:

- evaluate all of the arguments to the function in the current environment (from which the function is being called)
- make a new environment whose parent is the environment in which the function was *defined* (this is called lexical scoping)
- in that new environment, bind the function's parameters to the values that are passed to it
- evaluate the body of the function in that new environment

Examples

Here are two examples of using calling functions and using `lambda`. The first associates a name with the function and calls the function using that name, whereas the second calls the function directly without first giving it a name.

```
in> (define square (lambda (x) (* x x)))
    out> function object

in> (square 2)
    out> 4
```

```
in> ((lambda (x) (* x x)) 2)
    out> 4
```

Note that in either case, no binding for `x` is made in the original environment; this binding happens in the environment that is created when the function is called.

Here is another example of a more complicated function. Note that the result of calling `(foo 3)` is a *function*, which is then called with `2` as an argument. Note also that the value as sociated with the name `x` when we call `(bar 2)` is the `3` from the environment in which that function was *defined*, not the `7` from the environment in which it was called.

```
in> (define x 7)
    out> 7

in> (define foo (lambda (x) (lambda (y) (+ x y))))
    out> function object

in> (define bar (foo 3))
    out> function object

in> (bar 2)
    out> 5
```

Changes to `evaluate`

We will need to make sure that `evaluate` handles the `lambda` keyword properly, by creating a new function object that stores the names of the parameters, the expression representing the body of the function, and the environment in which the function was defined. We also need to modify evaluate to handle *calling* user-defined functions.

From a high-level perspective, your evaluator should now work in the following way, given an expression `e`:

- If `e` represents a number, it should evaluate to that number
- If `e` represents a special form (such as `define`), it should be evaluated acfording to the rules for that special form.
- If `e` represents a variable name, it should evaluate to the value associated with that variable in the given environment, or it should raise an `EvaluationError` if a binding cannot be found according to the rules above.
- Otherwise, `e` is a compound expression representing a function call. Each of the subexpressions should be evaluated in the given environment, and:
  - If the first subexpression is a built-in function, it should be called with the remaining subexpressions as arguments (in order).
  - If the first subexpression is a user-defined function, it should be called according to the rules given above.

After you have made the changes above, try them out in the REPL. Once you are reasonably certain they are working, try them with `test.py`. At this point, your code should pass tests 1-11 in `test.py`.

Easier Function Definitions

Implementing user-defined functions has given a lot of power to our interpreter! But it is kind of a pain to type them out. Implement a shorter syntax for function definitions, so that, if the `NAME` in a `define` expression is itself an S-expression, it is implicitly translated to a function definition before binding. For example:

- `(define (five) (+ 2 3))` should be equivalent to `(define five (lambda () (+ 2 3)))`
- `(define (square x) (* x x))` should be equivalent to `(define square (lambda (x) (* x x)))`
- `(define (add2 x y) (+ x y))` should be equivalent to `(define add2 (lambda (x y) (+ x y)))`

This is nice not only because it is easier to type, but also because it makes the definition of a function more closely mirror the syntax we will use when calling the function.

After implementing this change, try it out in the REPL, and then in `test.py`. At this point, your code should pass tests 1-21 in `test.py`.

Evaluator 4: Conditionals

Now we'll add support for conditional execution via the `if` special form, which has the following form: `(if COND TRUEEXP FALSEEXP)`

To evaluate this form, we need to first evaluate `COND`. If `COND` evaluates to true, the result of this expression is the result of evaluating `TRUEEXP`; if `COND` instead evaluates to false, the result of this expression is the result of evaluating `FALSEEXP`. Note that we should *never* need to evaluate both `TRUEEXP` and `FALSEEXP` when evaluating an `if` expression (for this reason, we cannot implement `if` as a function; it *must* be a special form).

Booleans and Comparisons

In order to implement `if`, we will need a way to represent Boolean values in *carlae*. This decision is up to you, but no matter your choice of representation, you should make these values available inside of *carlae* as literals `#t` and `#f`, respectively. We will also need several additional functions, all of which should take arbitrarily-many arguments:

- `=?` should evaluate to true if all of its arguments are equal to each other.
- `>` should evaluate to true if its arguments are in decreasing order.
- `>=` should evaluate to true if its arguments are in nonincreasing order.
- `<` should evaluate to true if its arguments are in increasing order.
- `<=` should evaluate to true if its arguments are in nondecreasing order.

As well as the following Boolean combinators:

- `and` should be a *special form* that takes arbitrarily many aguments and evaluates to true if *all* of its arguments are true. It should only evaluate the arguments it needs to evaluate to determine the result of the expression.
- `or` should be a *special form* that takes arbitrarily many arguments and evaluates to true if *any* of its arguments is true. It should only evaluate the arguments it needs to evaluate to determine the result of the expression.
- `not` should be a *function* that takes a single argument and should evaluate to false if its argument is true, and true if its argument is false.

After implementing these functions, modify your `evaluate` function so that it properly handles the `if` special form. Once you have done so, your code should pass all 34 tests in `test.py`.

With this addition, your interpreter should be able to handle recursion! Try running the following pieces of code from your REPL to check that this is working:

```
in> (define (factorial n) (if (<= n 1) 1 (* n (factorial (- n 1)))))
in> (factorial 6)
```

And feel free to play around and try some programs of your own!

Final Notes (for now!)

Congratulations; you've just implemented your first interpreter! By now your *carlae* interpreter is capable of evaluating arbitrarily-complicated programs (and in the second part of this lab, we will work on adding some improvements and new features to our *carlae* interpreter).

Hopefully this has been fun, interesting, and educational in its own right, but there are a few important reasons why we've chosen this as a project:

1. There is something powerful in understanding that an interpreter for a programming language (even one as complicated as Python) is *just another computer program*, and it is something that, with time and effort, you are capable of writing.

2. This is an example of a rather large and complicated program, but we were able to manage that complexity by breaking things down into small pieces.

3. Our little *carlae* interpreter actually has a lot in common with the Python interpreter, and so there is a hope that you have learned something not only about this little language, but also about how Python behaves. Among other things:

    - both run programs by breaking the process down into lexing, parsing, and evaluating, and
    - the way function calls are scoped and handled is very similar in the two languages,

4. Course 6 and LISP have a long history:

    - LISP was conceived by John McCarthy at MIT in 1958, and one of the most widely-used LISP dialects, Scheme, was developed here by Guy Steele and Gerry Sussman in 1970.
    - The predecessor to 6.009, 6.001 *Structure and Interpretation of Computer Programs*, was taught as part of the course 6 introductory series for around 30 years and used Scheme as its primary language. The associated text is still considered by many to be one of the best books ever written about computer programming.

Coding Points

There are 18 coding points in this lab:

- 4 will be awarded for "style," based on the structure and clarity of your code, and accompanying comments and docstrings.
- 9 will be awarded based on your implementation of Booleans, environments, and user-defined functions, including your choice of representation.
- 5 will be awarded for correctly implementing the REPL.