Lab 9: LISP (Part II) 🐍🐍

You are not logged in.

If you are a current student, please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

This lab assumes you have Python 3.6 or later installed on your machine (3.10 recommended).

The following file contains code and other resources as a starting point for this lab: `lab09.zip`

You should start by copying your `lab.py` file from Lab 8 into this week's distribution. Most of your changes will be made to this file.

You can also see and participate in online discussion about this lab in the "Labs" Category in the forum.

This lab is worth 4 points.

All non-checkoff questions on this page (including your code submission) are due at 5pm on Friday, 22 April 2022. However, you are *strongly* encouraged to start this lab early.

Your Checkoff This Week

You are not logged in. Please log in to see information about your checkoff for this lab.

Reminder: Academic Integrity

Please also review the academic integrity policies before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources.**

2) Introduction

In Lab 8, you implemented an interpreter for a dialect of LISP called *Carlae*. This lab builds on your work from Lab 8 to introduce some new features into the *Carlae* language, and we'll also ask you to implement a couple of small programs in *Carlae* yourself. If you have not yet finished Lab 8, you should do so before working on this lab.

Your code should pass the `test_oldbehaviors` test cases with no modification to your Lab 8 file. (Note, however, that we did add at least one new test, based on common student problems from Lab 8 that we thought would be good to help you catch before continuing!)

3) Conditionals

Our lab from last time could do some neat things, but it was still somewhat limited. One glaring deficiency is the lack of conditionals, so we'll start this week's portion by adding support for conditional execution via the `if` special form[1], which has the following form: `(if COND TRUE_EXP FALSE_EXP)`

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

Check Yourself:

Why is it important that `if` only evaluates one of the branches? Can you think of a situation where evaluating both branches would be problematic?

3.1) 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 built-in 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 arguments 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. For example, `(and (> 3 2) (< 7 8) @f)` should evaluate to false.
- `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. For example, `(or (> 3 2) @t (< 4 3))` should evaluate to true.
- `not` should be a *built-in function* that takes a single argument and should evaluate to false if its argument is true and true if its argument is false. For example, `(not (=? 2 3))` should evaluate to true. If `not` receives more than one argument, it should raise a `CarlaeEvaluationError`.

In *Carlae* (as in Python), `and` and `or` do not behave like functions, in the sense that they do not necessarily evaluate all their arguments; they only evaluate as far as they need to to figure out what their results should be (this kind of behavior is often referred to as "short circuiting").

Check Yourself:

`and` evaluates to true only if all of its arguments are true. So if we're evaluating all its arguments in order one-by-one, under what condition can we stop (and avoid evaluating the rest of the arguments)? What about `or`?

Check Yourself:

It could be nice to implement the comparison operations as special forms, too, so that they can short-circuit as well. If you want to do that, go ahead! But the test cases should pass either way.

After implementing these functions / special forms, modify your `evaluate` function so that it properly handles the `if` special form. Once you have done so, your code should pass the tests related to conditionals 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> (:= (fib n) (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2)))))
in> (fib 6)
```

3.2) Example Programs

Now is a good time for you to try your hand at writing a couple of small programs in *Carlae* itself (after all, an interpreter isn't much use without some programs to run!). In the boxes below, we'll check your code using our interpreter, but you should feel free to run some tests in your own interpreter as well!

3.2.1) Absolute Value

In the box below, enter a definition for the `abs` function, which takes a number n as input and returns $|n|$.

Note that, unlike your REPL, this box (and the others like it on this page) *do* accept multi-line expressions (you don't need to write everything on one line, although you are welcome to if you want).

Enter your definition below:

```
(:= (abs n)

)
```

3.2.2) Factorial

In the box below, enter a definition for the `factorial` function, which takes a nonnegative integer n as input and returns $n!$.

We'll test your code using our *Carlae* interpreter behind the scenes, but you can/should also feel free to test it on your own implementation!

Enter your definition below:

```
(:= (factorial n)

)
```

4) Lists

Next, we'll add support for *lists* (after all, despite the joke acronyms in the last lab, the name LISP is *actually* short for **LIS**t **P**rocessor, so we're not really done until we've got support for lists!). In *Carlae*, we will implement lists as linked lists (as is typical for LISP). While the high-level view will be the same as the linked lists we saw in recitation 10, the details here will be slightly different. The implementation is discussed in detail below.

4.1) Pairs

We'll start by implementing a built-in function called `pair`. `pair` exists to create ordered pairs, each of which consists of two values. We'll refer to these two values as the `head` and the `tail`, respectively.

You should start by implementing a class called `Pair` to represent a pair. Each instance of `Pair` should have ***exactly two* instance variables**:

- an attribute called `head` (which represents the first element in the pair), and
- an attribute called `tail` (which represents the second element in the pair).

You should also add support for the `pair` function to your interpreter. For example, calling `(pair 1 2)` should result in a new `Pair` object whose `head` is `1` and whose `tail` is `2`.

If `pair` is called with the wrong number of arguments, it should raise a `CarlaeEvaluationError`.

4.1.1) `head` and `tail`

In addition, implement two new built-in functions to retrieve the `head` and `tail`, respectively, from a given pair.

- `(head X)` should take a pair (an instance of your `Pair` class) as argument and should return the first element in the pair. If it is called on something that is not a pair, or if it is provided the wrong number of arguments, it should raise a

`CarlaeEvaluationError`.

- `(tail X)` should take a pair as argument and return the second element in the pair. If it is called on something that is not a pair, or if it is provided the wrong number of arguments, it should raise a `CarlaeEvaluationError`.

4.2) Lists

It may seem simple, but `pair` is a powerful tool. Among other things, `pair` can be used to implement *linked lists*. In this section, we'll add support for linked lists to our interpreter.

4.2.1) `nil` and Empty Lists

In some LISP dialects, a variable called `nil` (which is roughly equivalent to Python's `None`) is used to represent empty lists, and we will use that same convention for *Carlae*.

Add support for the word `nil` to your interpreter, so that evaluating `nil` produces some internal representation for `nil`.

For purposes of passing our test suite, the following things are important:

- Evaluating `nil` multiple times should always produce results that are equivalent to each other when comparing with `==`.
- Evaluating `nil` must produce a value that is distinct from `@f` and `0` (i.e., that is not equivalent to those things when comparing with `==`).
- `nil` should not be an instance of `Pair` (since it does not have a `head` or a `tail`).

4.2.2) Linked Lists

In most dialects of LISP, lists are implemented using pairs. In particular, a list is either:

- an empty list (represented by `nil`), or
- a pair cell whose `head` is the first element in the list and whose `tail` is a list containing the rest of the elements.

This forms the basis of a simple linked-list structure whose contents can be manipulated with `pair`, `head`, and `tail`. Note that `nil` is the only list that is not also a pair.

For example, consider the following examples:

- `nil` represents the empty list.
- `(pair 9 nil)` represents a list with a single element (`9`).
- `(pair 9 (pair 8 (pair 7 nil)))` represents a list containing the elements `9`, `8`, and `7`, in that order.

It's worth noting that, although lists are comprised of pairs, not all pairs are lists (for example, `(pair 1 2)` is not a list).

4.3) Built-in List Functions

In order to make using lists practical, we'll add support for more built-in functions designed to operate on lists.

To start, we'll add an easier way to create lists. It's kind of a pain to write a big chain of `pair` calls to create a new list. So we'll add support for the `list` function. This function should take zero or more arguments and should construct a linked list that contains those arguments, in order. You should make sure that calling `list` with no arguments produces our representation for an empty list.

For example:

- `(list)` should evaluate to the same thing as `nil`
- `(list 1)` should evaluate to the same thing as `(pair 1 nil)`
- `(list 1 2)` should evaluate to the same thing as `(pair 1 (pair 2 nil))`
- and so on.

In addition, we will define some additional built-in functions for operating on lists within *Carlae*.

All of the functions below should be implemented by operating directly on instances of `Pair` representing linked lists, without ever converting to or using a Python list/tuple.

Convenience Methods

Start by adding a few new built-in functions that that operate on lists:

- `(list? OBJECT)` should take an arbitrary object as input, and it should return `@t` if that object is a linked list, and `@f` otherwise.
- `(length LIST)` should take a list as argument and should return the length of that list. When called on any object that is not a linked list, it should raise a `CarlaeEvaluationError`.
- `(nth LIST INDEX)` should take a list and a nonnegative index, and it should return the element at the given index in the given list. As in Python, indices start from 0. If `LIST` is a pair (but not a list), then asking for index 0 should produce the `head` of that pair, and asking for any other index should raise a `CarlaeEvaluationError`. You do not need to support negative indices.
- `(concat LIST1 LIST2 LIST3 ...)` should take an arbitrary number of lists as arguments and should return a new list representing the concatenation of these lists. If exactly one list is passed in, it should return a copy of that list. If `concat` is called with no arguments, it should produce an empty list. Calling `concat` on any elements that are not lists should result in a `CarlaeEvaluationError`.

`map`, `filter`, and `reduce`

Beyond these functions, the following will allow us to easily construct new lists from existing ones:

- `(map FUNCTION LIST)` takes a function and a list as arguments, and it returns a *new list* containing the results of applying the given function to each element of the given list.

  For example, `(map (function (x) (* 2 x)) (list 1 2 3))` should produce the list `(2 4 6)`.

- `(filter FUNCTION LIST)` takes a function and a list as arguments, and it returns a *new list* containing only the elements of the given list for which the given function returns true.

  For example, `(filter (function (x) (> x 0)) (list -1 2 -3 4))` should produce the list `(2 4)`.

- `(reduce FUNCTION LIST INITVAL)` takes a function, a list, and an initial value as inputs. It produces its output by successively applying the given function to the elements in the list, maintaining an intermediate result along the way. This is perhaps the most difficult of the three functions to understand, but it may be easiest to see by example.

  Consider `(reduce * (list 9 8 7) 1)`. The function in question is `*`. Our initial value is `1`. We take this value and combine it with the first element in the list using the given function, giving us `(* 1 9)` or `9`. Then we take *this* result and combine it with the next element in the list using the given function, giving us `(* 9 8)` or `72`. Then we take *this* result and combine it with the next element in the list using the given function, giving us `(* 72 7)` or `504`. Since we have reached the end of the list, this is our final return value (if there were more elements in the list, we would keep combining our "result so far" with the next element in the list, using the given function).

The Wikipedia pages for map, filter, and reduce provide some additional examples.

If the arguments to `map`, `filter`, or `reduce` are not of the proper types, your code should produce a `CarlaeEvaluationError`.

Once we have these three functions, we have the equivalent of list comprehensions in *Carlae*! In Python, for example, we might write:

```
sum([i**2 for i in some_list if i < 0])
```

In *Carlae*, we can now do the same thing with the following code:

```
(reduce + (map (function (i) (* i i)) (filter (function (i) (< i 0)) some_list)) 0)
```

This is a lot to take in, but it gives us the same result:

- It first *filters* `some_list` to produce a list containing only the negative values.
- It then *maps* the `square` function onto the resulting list.
- Finally, it *reduces* that result by successive application of the `+` operator to produce the sum.

Implementation

Implement the `list`, `head`, `tail`, `length`, `nth`, `concat`, `map`, `filter`, and `reduce` functions and add them to the built-in functions. Once you have done so, your code should pass the test cases related to lists in `test.py`. **REMINDER that these functions should not operate by converting to Python lists/tuples.**

4.4) Example Programs

Now that we have lists available to us, it's again a good time to try your hand at writing some more *Carlae* code involving lists! Again, we'll write a couple of functions.

4.4.1) Home on the Range

First, let's implement a function to replicate the behavior of `range` in Python. In particular, define a function `(range start stop step)`, which outputs a list containing the same numbers that would exist in Python's `range(start, stop, step)`.

Importantly, unlike Python's `range`, you only need to handle the case of a positive `step` argument, and you can always assume that all three arguments will be provided.

Enter your definition below:

```
(:= (range start stop step)

)
```

4.4.2) Flatten List

Now, let's implement a function we've written a few times in Python this semester: a function that "flattens" a given list. Given a list that contains elements, some of which are lists (which could themselves contain other lists, etc), the `flat` function should return a single "flat" list, containing all of the elements from the original function but with all nesting of lists removed.

Enter your definition below:

```
(:= (flat L)

)
```

5) Evaluating Multiple Expressions

To help with the above, introduce a new built-in function called `begin`. `begin` should simply return its last argument. This is a useful structure for running commands successively: even though only the last argument is returned, all of the arguments are evaluated in turn, which allows us to run arbitrary expressions sequentially.

For example, `(begin (:= x 7) (:= y 8) (- x y))` should evaluate to `-1`.

After implementing `begin`, your code should pass `test_begin` (but not necessarily `test_begin2`, which depends on other pieces from later in the lab.

6) Reading From Files

OK, now that we have lists, conditionals, `map`, `filter`, `reduce`, and `begin`, we've got some *real* power. But it's kind of a pain to write even medium-sized programs with this infrastructure, since we are limited to evaluating one expression at a time via the REPL (and, even though our interpreter supports multi-line expressions, our REPL does not!).

In this section, we will get rid of this limitation by adding the capability to run the contents of a file before dropping into our REPL, which we can use, for example, to define multiple functions.

Define a function called `evaluate_file` in `lab.py`. This function should take a single argument (a string containing the name of a file to be evaluated) and an optional argument (the environment in which to evaluate the expression), and it should return the result of evaluating the expression contained in the file (you may assume that each file contains a single expression).

You may find the documentation for Python's built-in `open` function helpful.

At this point, your code should pass the tests in the tests related to files in `test.py`.

7) Command-Line Arguments

Now that we have the ability to evaluate the contents of a file in a particular environment, we will need to let Python know *which* files it should evaluate before dropping into the REPL. We will do this by passing the names of these files to *Carlae* as *command-line arguments*. For example, instead of just running:

```
$ python3 lab.py
```

we will run something like:

```
$ python3 lab.py some_definitions.carlae more_definitions.carlae
```

From inside of Python, these arguments are available as part of the `sys.argv` list (note that, if you haven't already, you should add `import sys` near the top of your file at this point). For the example above, `sys.argv` will contain:

```
['lab.py', 'some_definitions.carlae', 'more_definitions.carlae']
```

Modify `lab.py` so that, when `lab.py` is run with filenames passed in on the command line, it evaluates the expressions contained in those files into the global environment before entering the REPL. You may assume that each file contains only one expression. To test your code, you can make a few files that contain simple expressions you can check (for example, define a single variable inside a file and make sure it is available to you from the REPL after that file is evaluated).

After you have implemented `begin` and command-line arguments, you should be able to run `python3 lab.py test_files/definitions.carlae` to grab some definitions into the REPL.

8) Variable Binding Manipulation

We will finish by implementing a couple of additional special forms, which can be used to manipulate variable bindings: `del`, `let` and `set!`.

With these pieces implemented, we will have the ability to use object-oriented programming from within *Carlae*.

8.1) `del`

`del` is used for deleting variable bindings within the current environment. It takes the form: `(del VAR)`, where `VAR` is a variable name. If the given variable is bound in the current environment, its binding should be removed, and the associated value should be returned. If `VAR` is not bound locally, this special form should raise a `CarlaeNameError`.

Note that implementing this behavior correctly requires that your environments are structured as described in section 6.4.2 of Lab 8, i.e. that the built-ins are in a separate environment that is the parent of the global frame.

8.2) `let`

`let` is used for creating *local variable definitions*. It takes the form: `(let ((VAR1 VAL1) (VAR2 VAL2) (VAR3 VAL3) ...) BODY)`, where `VAR1`, `VAR2`, etc., are variable names, and `VAL1`, `VAL2`, etc., are expressions denoting the values to which those names should be bound. It works by:

- Evaluating all the given values in the current environment.
- *Creating a new environment* whose parent is the current environment, binding each name to its associated value in this new environment.
- Evaluating the `BODY` expression in this new environment (this value is the result of evaluating the `let` special form).

Note that the given bindings are only available in the body of the `let` expression. For example:

```
in> (:= z 5)
    out> 5

in> (let ((x 5) (y 3)) (+ x y z))
    out> 13

in> x
    EXCEPTION!

in> y
    EXCEPTION!

in> z
    out> 5
```

8.3) `set!`

`set!` (pronounced "set bang") is used for *changing the value of an existing variable*. It takes the form: `(set! VAR EXPR)`, where `VAR` is a variable name, and `EXPR` is an expression.

It should work by:

- Evaluating the given expression in the current environment
- Finding the nearest enclosing environment in which `VAR` is defined (starting from the current environment and working upward until it finds a binding) and updating its binding *in that environment* to be the result of evaluating `EXPR`

It should also evaluate to that same value.

If `VAR` is not defined in any environments in the chain, `set!` should raise a `CarlaeNameError`.

```
in> (:= x 7)
    out> 7

in> (:= (foo z) (set! x (+ z 2)))
    out> function object

in> (foo 3)
    out> 5

in> x
    out> 5

in> (:= (bar z) (:= x (+ z 2)))
    out> function object

in> (bar 7)
    out> 9

in> x
    out> 5
```

Implement `let` and `set!` in `lab.py`. After doing so, your code should pass all the tests in `test.py`! Note that the last 6 test cases are realistic programs implemented in *Carlae* (including implementations of N-D Minesweeper and a Sudoku solver)! The code for the N-D Minesweeper implementation and the Sudoku solver are also available in the `test_files` directory in a format that makes them a little easier to read, in case you want to look at them, try them in your REPL, or modify them.

9) Endnotes and Commentary

Congratulations; you've just implemented your first interpreter! By now your *Carlae* interpreter is capable of evaluating arbitrarily complicated programs!

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. We wanted to give you an opportunity to make some more open-ended design decisions than you have been asked to make in the past, and this lab offers such an opportunity.

4. 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.

5. 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 Jerry 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 textbook is still considered by many to be one of the best books ever written about computer programming.

Now that we have a working interpreter, Section 12 includes a few suggestions of neat (but optional) additional features that you might consider adding to your interpreter. Although these are not required, they would make great extra practice if you're looking for it!

10) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `submit-009-lab` script. The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ submit-009-lab -a lab09 /path/to/lab.py
```

Running that script should submit your file to be checked, and it should also provide some information about how and where to get feedback about your submission. Reloading this page after submitting will also show some additional information:

> You have not yet made any submissions to this assignment.

11) Checkoff

Once you are finished with the code, please come to office hours and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code and test cases in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions). In particular:

- Did the additions in this lab require you to rethink any of your design choices from lab 8? If so, please briefly describe those changes.
- `nil` could be represented a number of different ways, including: the string `"nil"`, Python's `None` object, the `False` Boolean, `0`, or a custom class (or instance thereof). What would be some pros and cons of each of these choices? If you chose a different representation than one of these, include your representation in the comparison.
- Briefly describe how you implemented short-circuiting in `and` and `or`. Why is short-circuiting desirable?
- Briefly explain your general strategy for the various linked-list functions (without converting to Python lists/tuples).
- Briefly explain `let` and `set!`. What are the differences between these two special forms?
- Explain how your interpreter will tokenize, parse, and evaluate a small expression like the following, assuming your definition of `factorial` from above is used:

  ```
  (begin
      (:= x 7)
      (:= y 9)
      (:= (square x) (* x x))
      (factorial (square 2))
  )
  ```

  How many times is `evaluate` or an associated helper called? What are they called on? How many environments are made in total? What names are bound in each environment?

> *You have not yet received this checkoff.*

12) Optional Improvements and Extensions

12.1) Turtle Graphics

We've done some really neat things so far, but, as of right now, the kinds of programs we can actually write is somewhat limited by the fact that we can really only interact with numbers inside of our *Carlae* programs. We can resolve this in myriad ways (such as including strings in our language). But one neat way to expand the capabilities of our interpreter is to add support for "turtle graphics", which provide an intuitive way to create interesting graphics (it turns out that from this simple model, we can create some really interesting pictures!).

We have provided a file called `cturtle.py` in this week's distribution, which provides a small wrapper around Python's own built-in `turtle` module used for interactively drawing pictures. We can add support for turtle graphics in *Carlae* by hooking into this module.

In particular, if you want to try this, you can add the following to the top of your `lab.py` file:

```
from cturtle import turtle
```

And add another special form to your language:

```
(turtle COMMAND ARGUMENTS)
```

where `COMMAND` is a command to give to the imaginary turtle that is doing our drawing, and `ARGUMENTS` are the arguments to that command. You can implement this by invoking the imported `turtle` function directly. For example, evaluating `(turtle forward 100)` should ultimately call the following from within Python: `turtle("forward", [100])`.

We have made the following commands available to you, some examples below:

- `(turtle forward 100)` moves the turtle forward by 100 pixels, in a straight line (we can move backward by providing a negative number).
- `(turtle left 60)` rotates the turtle, in place, by 60 *degrees* to its left (we can turn right by providing a negative number).
- `(turtle penup)` lifts the turtle's "pen" off of the paper (so that moving the turtle does not draw a line).
- `(turtle pendown)` puts the pen down (so that moving the turtle will draw a line).

We have also provided some methods for querying or setting the turtle's position and heading.

Some sample programs making use of the turtle graphics are provided in the `turtle_samples/` directory of the code distribution to give you a sense of some of the neat pictures we can draw here. You should be able to run them, for example, with:

```
python3 lab.py turtle_samples/tree.carlae
```

Of course, after having looked at those, you might find it interesting to write some programs of your own that generate other interesting pictures!

Note: if you submit your lab with the `turtle` special form enabled, you need to comment out the import statement above or wrap it in a `try-except` block as below, otherwise it will cause errors.

```
try:
    from cturtle import turtle
except Exception:
    turtle = None
```

12.2) Tail-Call Optimization

If you have the time and interest, a *really* interesting and powerful optimization for our interpreter comes in the form of *tail-call optimization*.

A typical way of structuring the evaluator from above involves making recursive calls to `evaluate`, in particular when calling functions. This is a nice way of structuring things, but it actually leads to issues. For example, try calling `(fib 2000)` from your REPL after loading in the definitions from `test_files/definitions.carlae`. What happens?

We run into issues with recursive calls because Python (necessarily) has a limit on the "depth" it will allow in a recursive call, to prevent infinite recursion. Even if Python didn't have this limit, it would end up using quite a lot of memory allocating new stack frames for these recursive calls.

A neat optimization to avoid this problem is to implement tail-call optimization, whereby we can avoid some of these issues (allowing, for example, computing `(fib n)` for arbitrarily large `n`).

In short, many pieces of our interpreter involve returning the result of a single new call to `evaluate`, with a different expression and a different environment. In those situations (conditionals, calling user-defined functions, etc.), it would be much better from an efficiency perspective to avoid the recursive call to `evaluate`; rather, we can simply adjust the values of the expression and the environment within the same call to `evaluate` by introducing a looping structure: keep looping until we have successfully evaluated a structure, and if the result is simply the result of evaluating another expression (potentially in a different environment), then adjust those values as necessary.

There are no tests for this optimization, but after doing so, your code should work for `(fib 100000)` (or arbitrarily high `n`)!

12.3) Additional (Optional) Improvements and Exercises

If you have the time and interest, you can improve upon this base language by implementing some additional features. Below are some ideas for possible improvements, or just for ways to get extra practice. These are by no means required, but we are still happy to help with them if you get stuck!

- If you haven't already done so, add syntax checking (like we did in lab 8) for the new special forms introduced in this lab.
- Add support for multi-line expressions to your REPL. If a user enters something that could be a valid start of an expression (but is not a complete expression), you should continue prompting for input until the result forms a valid expression (in which case you should evaluate it) or something that could not be the start of a valid expression (in which case you should report a syntax error).
- To support print-statement debugging, add a built-in function `display`, such that `(display EXPR)` prints the result of `EXPR` and then returns it.
- Allow the body of a function defined with the `function` special form, or the body of a `let` expression, to consist of more than one expression (this should be implicitly translated to a `begin` expression).
- Add a function called `set-head!` that changes the first element of a pair to a particular value. Then use your list built-ins to implement a `set-nth` function in *Carlae*.
- Add support for the `quote` and `eval` special forms. `(quote EXPR)` should return the given expression *without evaluating it*. Passing such an expression into `eval` should that evaluate it. For example, running `(eval (quote a))` should give the same result as evaluating `a`.
- Implement strings as an additional data type (be careful of how you handle comments to make sure that a `;` within a string doesn't get treated as the start of a comment!).
- Improve the system's error reporting by providing meaningful error messages that describe what error occurred.
- Since *iteration* doesn't exist in *Carlae* (except via recursion), try implementing some simple programs in *Carlae* for extra practice with recursion!
- Add support for importing functions and constants from Python modules (by mapping *Carlae* functions to the the `__import__` and `getattr` Python functions) so that the following will work:

```
in> (:= math (py-import (quote math)))
in> (:= sqrt (getattr math (quote sqrt)))
in> (sqrt 2)
    out> 1.4142135623730951
```

Footnotes

[1] Recall that a "special form" is an S-expression that doesn't follow the rules of a regular function call.