Lab 7: carlae

The questions below are due on Friday April 24, 2020; 04:00:00 PM.

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.csail.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.

The following file contains code and other resources as a starting point for this lab: `lab7.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

You can also see and participate in online discussion about this lab in the "Lab 7" Category in the forum.

This lab is worth a total of 4 points. Your score for the lab is based on:

- correctly answering the questions throughout this page (0.5 points)
- passing the test cases from `test.py` under the time limit (1 points), and
- a brief "checkoff" conversation with a staff member to discuss your code (2.5 points).

All questions on this page (including your code submission) are due at 4pm EDT on Friday, 24 Apr. However, you are *strongly* encouraged to start this lab early.

Note that no solutions will be published for lab 7, until after lab 8 has also come due.

Please also review the collaboration policy 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.**

NOTE that the test cases (`test.py`, `test_inputs/` and `test_outputs/`) have changed since the lab was released. An updated code distribution was released at around 9pm EDT on Friday, 17 April. If you downloaded the code distribution before that time, please re-download to get the updated test code. This distribution contains three additional test cases designed to help catch additional common errors (the correct test suite has 29 tests, not 26).

2) Introduction

Throughout 6.009, we have spent a lot of time discussing the behind-the-scenes operation of Python through environment diagrams. But you may have wondered along the way: how does Python actually *do* those things? *What* is actually doing them? It turns out that the thing doing that work is another program! In particular, a program called CPython, the Python interpreter, is responsible for interpreting your programs' source code and actually implementing the corresponding behaviors.

In this lab, we'll explore this idea even further by implementing our own 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 *Tetracheilostoma 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 powerful! We will end up implementing a large portion of the environment model we have been discussing, and our little language will actually 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 write *carlae* implementations of any of the labs we have done so far in 6.009!).

2.1) 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:

- Numbers (e.g., `1`) and symbols (things like variable names, 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 one or more expressions, followed by a closing round bracket `)`. The first subexpression determines what the S-expression means:
  - An S-expression starting with a keyword, e.g. `(define ...)`, is a *special form*; the meaning depends on the keyword[1].
  - 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 two bullet points above. For example, consider the following definition 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)))
    )
  )
)
```

Here is a brief example of an interaction with a LISP interpreter that demonstrates a few more examples of the structure of the language:

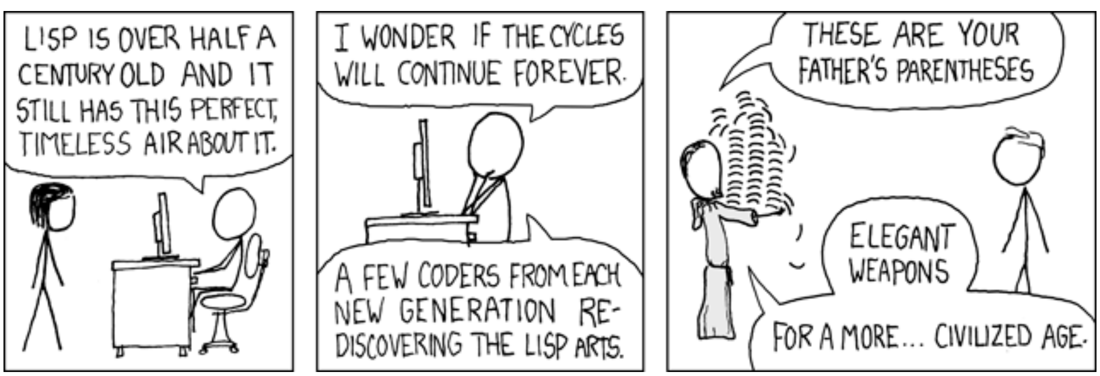
```
> (+ 3 2)

  => 5
```

```
> (define (square x) (* x x))

  => FUNCTION: (lambda (x) (* x x))


> (define (fourthpower x) (square (square x)))

  => FUNCTION: (lambda (x) (square (square x)))


> (fourthpower 1.1)

  => 1.4641000000000004


> (+ 3 (- 7 8))

  => 2
```

There are a couple of key syntactic differences from Python:

- LISP consists only of expressions, with no statements. Every expression returns a value.
- LISP uses prefix notation, e.g., `(+ 3 2)` instead of `3 + 2`.
- LISP's syntax is simpler, but consists of a lot more parentheses.

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." `:)`



from XKCD

3) 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, our *carlae* interpreter will consist of three parts:

- A *tokenizer*, which takes a string as input and produces a list of *tokens*, which represent meaningful units in the syntax of the programming language.
- A *parser*, which takes the output of the tokenizer 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.

4) Tokenizer

Our first job is *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', ')', ')']`. Note that at this point, all of our tokens should be strings.

Unlike in Python, indentation 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. A comment in *carlae* is 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.

What should be the result of `tokenize("(cat (dog (tomato)))")`?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

What should be the result of tokenizing the following expression?

```
;add the numbers 2 and 3
(+ ; this expression
 2     ; spans multiple
 3  ; lines

)
```

This question is due on Friday April 24, 2020 at 04:00:00 PM.

Implement the `tokenize` function in `lab.py`. Note that, while doing so, you may find some of the methods of the built-in `str` type useful when doing so.

5) 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 according to its Python type (i.e., integers as `int` and decimals as `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']]]]
```

If the parentheses are mismatched in the input, or if you are given tokens that otherwise don't represent a single valid expression as described above, `parse` should raise a `SyntaxError`.

After implementing both `tokenize` and `parse`, your code should pass the `Test1_Parse` suite in `test.py`. That said, you should read this whole section before jumping in to implementation.

5.1) Examples

What should be the result of calling `parse` on the output from the first concept question above?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

What should be the result of calling `parse(['2'])`?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

What should be the result of calling `parse(['x'])`?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

What should be the result of calling `parse(['(', '+', '2', '(', '-', '5', '3', ')', '7', '8', ')'])`?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

5.2) Parsing Algorithm

One way to structure things, from a high-level, is to use a *recursive descent* parser. From a high level, we can structure parse with something like the following:

```
def parse(tokens):
    def parse_expression(index):
        pass  # your code here
    parsed_expression, next_index = parse_expression(0)
    return parsed_expression
```

The function `parse_expression` is a recursive function that takes as argument an integer into the `tokens` list and returns a pair of values:

- the expression found starting at the location given by `index`, and
- the index beyond where this expression ends (i.e., if the expression ends at the token with index `6` in the `tokens` list, then the returned value should be `7`).

In the definition of this procedure, we make sure that we call it with the value `index` corresponding to the start of an expression. So, we need to handle three cases. Let `token` be the token at location `index`; the cases are:

- **Number**: If `token` represents an integer, then we can just return that number (as an `int` or `float`, **not** as a string), paired with `index + 1` (since a number is represented by a single token).

- **Variable**: If `token` represents a symbol (a variable name), then we can just return that, paired with `index + 1` (since a symbol is represented by a single token).

- **S-expression**: Otherwise, the sequence of tokens starting at `index` must be an S expression containing one or more subexpressions, `(E1 E2 E3 ...)`. Since our S-expressions always start with parentheses, `token` must be `(`. In this case, we need to recursively parse each of the subexpressions, combine them into an appropriate representation of the S-expression overall, and and return that instance, along with the index of the token beyond the final right parenthesis.

Some questions that might be worth considering:

- how can we parse the first subexpression?
- assuming that succeeded, how can we parse the second subexpression?
- how do we know when we have parsed all of the subexpressions?

We have intentionally left out some of the details here (including how to check for syntax errors), but we think the above structure can provide a good starting point. If you get stuck, please don't hesitate to ask during office hours or via the forum.

Your code in this lab (including `parse`) should not use Python's built-in `eval` or `exec` functions.

6) Evaluator

> *"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 (maybe every time you get some new test cases to pass), so that if something goes wrong, you can revert to a working copy. However, note that you are not allowed to use a public code forge like Github or Gitlab for your backups.

6.1) 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. Look at the `carlae_builtins` dictionary to get a sense of the form of those functions.

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 representing 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 as arguments[2]. 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`, it should raise a `NameError`. Or, if it is a list whose first

element is not a valid function, it should raise an `EvaluationError`[3].

For example:

- `evaluate('+')` should return the function object associated with addition.
- `evaluate(3.14)` should return `3.14`.
- `evaluate(['+', 3, 7, 2])`, which corresponds to `(+ 3 7 2)`, should return `12` (i.e., the result of calling the `+` function on the given arguments).

Note that this should work for nested expressions as well. `evaluate(['+', 3, ['-', 7, 5]])`, which corresponds to `(+ 3 (- 7 5))`, should return `5`.

Implement the `evaluate` function in `lab.py` according to the rules above.

6.1.1) Testing Your Code: 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). If an exception occurs during that evaluation, you should report the error but your REPL should not stop (i.e., it should move on to asking for the next expression to be evaluated rather than erroring out).

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
```

Note that your REPL only needs to handle one-line inputs. You do not need to handle the case of a multi-line expression being entered through the REPL.

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.

6.2) 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[4].

(*Hint:* where can you put these operations so that they are usable by the interpreter?)

After implementing the evaluator and the `*` and `/` operations, try them out in the REPL.

6.3) Evaluator 2: Variables

Next, we will implement our first special form: *variable definition* using the `define` keyword (you may wish to read this section and section 6.4 in their entirety before implementing this behavior).

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 name that follows it; it only evaluates the expression that 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*.

6.4) 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 "built-in" 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 environment to the primitive addition procedure we defined above. The global environment will have this environment as its parent.

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, a `NameError` 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.

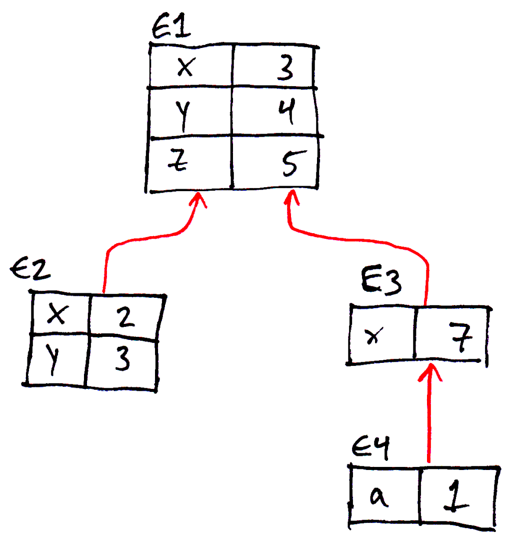
In order to make variables work properly, you will need to implement the kind of lookup described above in Python. To this end, **you should create a class to represent environments**.

> **Check Yourself:**
>
> What information should be stored as instance attributes of this class? What kind of methods should the class support?

It is up to you to decide how exactly to implement environments and the associated lookups within this structure. Details of your implementation will not be tested directly by the automatic checker; rather, your implementation will be tested by looking at the end-to-end behavior of your evaluator. Regardless of the details of your implementation, **you should make sure your environment representation can handle variables with arbitrary names** (i.e., any sequence of characters that doesn't represent a number and that doesn't contain parentheses or spaces should be treated as a variable name), and you should be prepared to discuss your implementation during the checkoff.

6.4.1) 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 (each of which might be an instance of the class you created to represent 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 environment is crucial for determining the result of an expression.

What is the result of evaluating `x` in **E1**?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

What is the result of evaluating `x` in **E2**?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

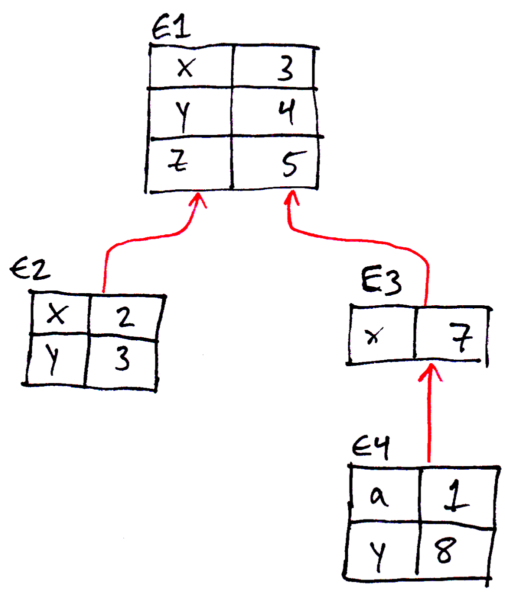
What is the result of evaluating `x` in **E3**?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

What is the result of evaluating `x` in **E4**?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

Also note that the `define` keyword always operates directly on the environment in which it is evaluated. 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):



Answer the following questions about variable lookup in this updated environment:

What is the result of evaluating `y` in **E1**?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

What is the result of evaluating `y` in **E2**?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

What is the result of evaluating `y` in **E3**?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

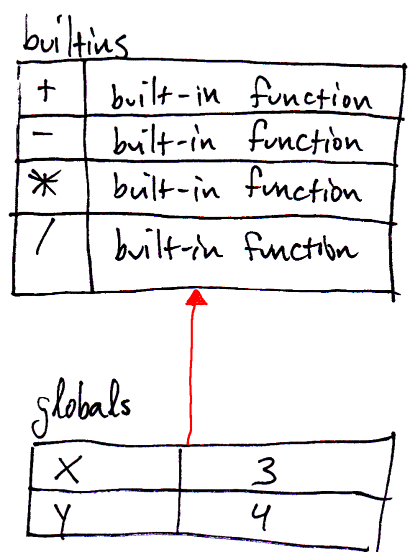
What is the result of evaluating `y` in **E4**?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

6.4.2) 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
```



Note that, when we look up any of the built-in variables from the global environment, we end up finding them in the built-ins environment.

6.5) Evaluator Changes

Now, we'll need to add support for variable definition and lookup to our interpreter by implementing a Python structure for representing an environment and modifying the `evaluate` function so that it handles variables and the `define` keyword.

Beyond implementing a representation for environments, we will need to make some 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 ultimately evaluated. **Your code will not pass the tests without this function.** Note that this function should always return the environment in which the expression was actually evaluated, *even if no environment was explicitly passed to it*.

Consider evaluating the following expressions in order from the same REPL (in the same environment). What is the result of evaluating each of these expressions?

input: `(+ 3 2 4 (- 2 7 8))`
output:
This question is due on Friday April 24, 2020 at 04:00:00 PM.

input: `(define x (+ 2 3))`
output:
This question is due on Friday April 24, 2020 at 04:00:00 PM.

input: `(define x2 (* x x))`
output:
This question is due on Friday April 24, 2020 at 04:00:00 PM.

input: `(+ x2 x)`
output:
This question is due on Friday April 24, 2020 at 04:00:00 PM.

Note that these can make for some nice test cases when you implement your code! You may also wish to test some other similar examples in your REPL.

At this point, your code should pass all the tests in the `Test2_Eval` suite in `test.py`.

Note also that, for all of the test cases moving forward, there is a corresponding file in the `carlae_code` directory; these files contain the expressions that are evaluated in the associated test case, with one expression on each line. A good exercise if you are failing a test case would be to figure out by hand what each expression should evaluate to and then try them by pasting one line at a time into your REPL. If you are having trouble figuring out why a given expression evaluates the way it does, please don't hesitate to ask for help in office hours or on the forum!

6.6) 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*.

6.6.1) 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

You should define a class to represent your user-defined functions. Once again, it is up to you to determine how exactly a function should be represented; 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 types of objects we have seen so far)[5].

Check Yourself:

What information should be stored as instance attributes of this class? What kind of methods should the class support?

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.

6.6.2) 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 arguments that are passed to it
- evaluate the body of the function in that new environment

If we try to call something that is not a function, or if we try to call a function with the incorrect number of arguments passed in, an `EvaluationError` should be raised.

It is worth noting that these steps are very close, if not exactly the same, as the steps Python takes when it calls a function. The readings for 6.145 contain some additional explanation and examples (see sections 4-8 of Functions and sections 1-2.5 of Functions 2).

6.6.3) Examples

Here are two examples of 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)) 3)
    out> 9
```

After evaluating the above code, what is the value of the variable `x` in the global environment?

--

This question is due on Friday April 24, 2020 at 04:00:00 PM.

Here is another example of a more complicated function. Note that the result of calling `(foo 3)` below is a *function*, which is then called with `2` as an argument. Note also that the value associated 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)
```

What is the result when evaluating `(bar 2)` above?

This question is due on Friday April 24, 2020 at 04:00:00 PM.

After evaluating the above code, what is the value of the variable `x` in the global environment?

--

This question is due on Friday April 24, 2020 at 04:00:00 PM.

After evaluating the above code, what is the value of the variable `y` in the global environment?

--

This question is due on Friday April 24, 2020 at 04:00:00 PM.

Check Yourself:

Why does `(bar 2)` not evaluate to `9`?

Show/Hide

6.6.4) 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 variable name, it should evaluate to the value associated with that variable in the given environment, or it should raise an `NameError` if a binding cannot be found according to the rules above.
- If `e` represents a special form (such as `define`), it should be evaluated according to the rules for that special form.
- 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.

If we try to call something that is not a function, or if we try to call a function with the incorrect number of arguments passed in, an `EvaluationError` should be raised.

NOTE that in the case of an S-expression that isn't a special form, evaluating the first element in the S-expression gives us the function that is to be called, regardless of how it is specified (so your evaluator code should not have additional logic based on the syntactic form of that first element).

After you have made the changes above, try them out in the REPL using the examples from subsubsection 6.6.3. Once you are reasonably certain that everything is working, try them with `test.py`. At this point, your code should pass the tests in the `Test3_Func` suite in `test.py`.

6.6.5) 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.

Modify your `evaluate` function so that it handles this new form. After implementing this change, try it out in the REPL, and then in `test.py`. At this point, your code should pass all the tests in `test.py`.

7) Endnotes

At this point, we have a very nice start toward an interpreter for *carlae*. We have the ability to define variables, and to define and call functions. Note also that recursion and closures come about naturally as a result of the rules we have implemented for calling functions (we don't have to do any additional work to get those features!). And hopefully implementing it has been an illuminating experience!

However, we still have some work to do before our language is complete. In particular, next week, we will add support for conditionals and lists.

8) Code Submission

Select File No file selected

This question is due on Friday April 24, 2020 at 04:00:00 PM.

9) Checkoff

Once you are finished with the code, please come to a tutorial, lab session, or office hour 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, be prepared to discuss:

- your implementation of `tokenize` and `parse`
- the structure of your class for representing environments
- how your environment representation handles arbitrary variable names
- how you set up and use environments in `evaluate` and `result_and_env`
- the structure of your class for representing user-defined functions
- how user-defined functions are called (including setting up the proper new environments)
- a demonstration of your REPL (including shared environment, exiting on QUIT, staying alive after errors, and correctly evaluating input expressions)

9.1) Grade

You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.

Footnotes

[1] For this lab, we will only have two keywords: `lambda` and `define`. In next week's lab, though, we will add more keywords and special forms.

[2] Note that we pass these arguments to the function as a single list containing all arguments, rather than as separate arguments

[3] Note that when you raise an exception, you can include an error message with details about the specifics of that error with the following syntax: `raise NameError("some error message here")`. This can help greatly with debugging, as it can help you locate the particular `raise` statement an error came from (and, ultimately, the cause of the exception).

[4] Although we will not be explicitly testing for them, there are a few "edge cases" to consider when implementing these functions. For multiplication, if we take the product of some numbers $a_0 \times a_1 \times a_2 \times \ldots$, and multiply it by the product of some numbers $b_0 \times b_1 \times b_2 \times \ldots$, we should get the same result as if we multiplied $a_0 \times a_1 \times a_2 \times \ldots \times b_0 \times b_1 \times b_2 \times \ldots$. This suggests:

- Multiplication with 1 argument passed in should return that argument itself
- Multiplication with no arguments should return 1

(In that same vein, note that `(+)` evaluates to `0`, and `(+ 2)` evaluates to `2`)

For division, it's not clear that such a relationship exists. But one set of rules that could make sense (and which is implemented in the MIT/GNU Scheme dialect of LISP) is:

- Division with no arguments should raise an exception.
- Division with a single argument x should return $1/x$.

[5] If you want to, you can implement the `__call__` "dunder" method to make instances of your class be callable like regular Python functions.