LISP Interpreter, Part 1

Table of Contents

1) Preparation

This lab assumes you have Python 3.11 or later installed on your machine (3.12 recommended).

The following file contains code and other resources as a starting point for this lab: `lisp_1.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.

Your raw score for this lab will be out of 5 points:

- answering the questions on this page (0.5 points),
- passing the server style check (1 point), and
- passing the test cases in `test.py` (3.5 points).

> **Please Read!**
>
> This is the first part of a two-part lab. Most of the test cases for next week's lab will require all of Lisp Part 1 to work. Because of this, you are **strongly** encouraged to start this lab early if possible. This first part is due 5:00pm on Friday, 03 May.

> **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, AI, and advanced code-completion tools.**

2) Introduction

Throughout 6.101, 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 and one with a strong historical connection to MIT (LISP was invented by John McCarthy at MIT in 1958, and the Scheme dialect of LISP was used in MIT's introductory programming subject from ~1975-2007[1]!).

The LISP dialect we'll implement in this lab (a variant/subset of Scheme) will be similar to Python in many ways, but in some ways it will be a bit smaller. 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 use it to write implementations of any of the labs we have done so far in 6.101!).

2.1) LISP and Scheme

As with most LISP dialects, the syntax of Scheme is far simpler than that of Python. All-in-all, we can define the syntax of Scheme 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 Scheme, 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 the S-expression means:
  - An S-expression starting with a keyword, e.g. `(lambda ...)`, is a *special form*; the meaning depends on the keyword. We will implement several special forms in this lab, including `lambda` (for creating function objects) and `define` (for defining variables).
  - An S-expression starting with a non-keyword, e.g. `(fib ...)`, is a function call, where the first element in the expression is the function to be called. Note that the first element can be an arbitrary expression. That is, it does not need to be a single name; it could be a more complicated expression that evaluates to a function object.

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 Scheme:

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

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

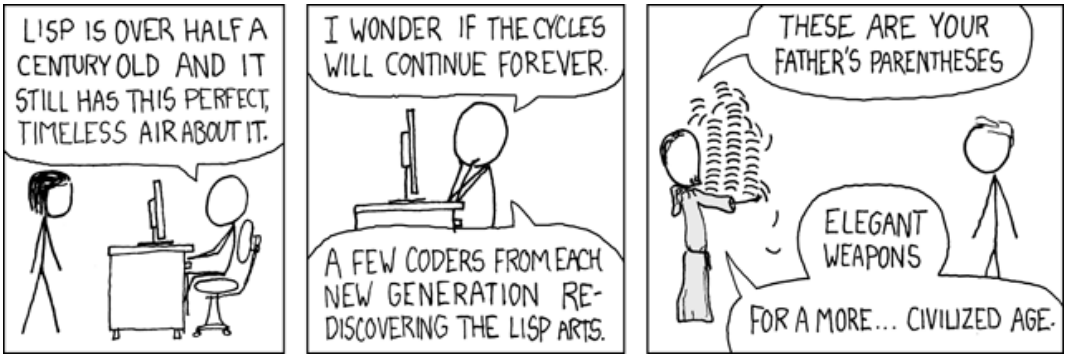
```
> (+ 3 2)

  => 5
```

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

> (define (fourthpower x) (square (square x)))
  => (lambda (x) (square (square x)))

> (fourthpower 1.1)
  => 1.4641000000000004

> (+ 3 (- 7 8))
  => 2
```

There are a couple of key syntactic differences from Python:

- Scheme consists only of expressions, with no statements. Every expression evaluates to a value.
- Scheme uses prefix notation, e.g., `(+ 3 2)` instead of `3 + 2`.
- Scheme'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 Scheme 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 Scheme 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 Scheme, we'll have exactly three kinds of tokens: opening round brackets `(`, closing round brackets `)`, and everything else. Each token is separated by whitespace (either spaces or the special "newline" character `'\n'`). 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 Scheme is prefixed with a semicolon (`;`). 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. Note that lines are separated by a special character `'\n'`.

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

What should be the result of tokenizing the following expression?

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

)
```

Check Yourself:

Note that this is similar in many ways to the tokenizer we wrote as part of the symbolic-algebra lab. What are the key similarities and key differences? How can we modify that tokenizer to work in this context?

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.

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']]]]
```

Note that the syntax of this language is similar to that of the little language we created for symbolic algebra in the previous lab, and so it might be a good idea to structure your parser for this lab in a similar fashion. One key difference between the two languages is that, while our symbolic-algebra language always had exactly two subexpressions to parse inside of parentheses, S-expressions can contain arbitrarily many subexpressions. Another is that we will handle malformed input graciously; in particular, in the case where

an expression is not well-formed, the function should raise a `SchemeSyntaxError`.

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

Note that we have also provided a function `number_or_symbol` in `lab.py`, which you can use to check whether a given string represents a number and to convert to the appropriate type if necessary. This function may be useful when thinking about differentiating between symbols and numbers in your parser.

You may also find the following questions helpful before jumping into your implementation (and you may even want to encode these, or similar examples, as doctests in your code).

5.1) Examples

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

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

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

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

6) Evaluator

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

After parsing, we have the program in a form that is (relatively) easy to work with, and 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 useful to save backups of your `lab.py` file after every major modification (maybe every time you get some new test cases to pass or every time you finish a new piece of functionality), so that if something goes wrong, you can revert to a working copy.

Note that you can also submit your lab code as many times as you like and that you can use this page to access files that you had previously submitted, so that can serve as another way to back up your work.

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 `scheme_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 `scheme_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, it should return the object associated with the symbol in `scheme_builtins`. If the symbol is not in `scheme_builtins`, it should raise a `SchemeNameError`.
- 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 list whose first element evaluates to something that is not a callable function, it should raise a `SchemeEvaluationError`[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've provided a REPL (a "**R**ead, **E**valuate, **P**rint **L**oop") for Scheme in your `lab.py` file. 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.

The REPL is implemented in a rather-obtuse class called `SchemeREPL` in your `lab.py` file. You don't have

to worry too much about this code if you don't want to; its job is to provide a nice interface for interacting with our interpreter. Here is one sample interaction with the REPL:

```
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 the REPL only handles one-line inputs.

By default, if your code raises a `SchemeError` (or an instance of one of its subclasses), the REPL will display some information about that exception but not exit. While this is the usual behavior we might expect from a REPL, it can be useful to see more information (such as a traceback) while debugging. To that end, if you change the value associated with `verbose` in the last line of `lab.py` from `verbose=False` to `verbose=True`, the REPL will show a full traceback, which can be a useful way to figure out what line is causing the error. While the traceback is helpful, we strongly encourage you to include descriptive messages when you raise errors i.e. `raise SchemeEvaluationError(f"Wrong number of arguments provided to custom function")`

Moving forward, we expect that **the REPL can and should be one of your main means of testing**; feel free to try things out using the REPL as you work through the remainder of the lab (by running the code from our test cases and seeing its output and/or by writing code of your own!). Importantly, when you make changes to your `lab.py` file you should save it and restart the REPL in order to observe what effect the changes made.

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) Adding Support for 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 Scheme 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`[5].

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 *frames*.

6.4) Frames

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 *frames* (which should be familiar from our environment diagrams). A frame consists of bindings from variable names to values, as well as possibly a parent frame, from which other bindings are inherited. One can look up a name in a frame, and one can bind names to values in a frame.

The frame 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 meanings. Rather, an expression acquires a meaning only with respect to some frame 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 frame*.

To describe interactions with the interpreter, we will suppose that there is a "built-in" frame, 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 frame to the primitive addition procedure we defined above. The global frame will have this frame as its parent.

One necessary operation on frames 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 frame, that value is returned.
- If the name does not have a binding in the frame and the frame has a parent, we look up the name in the parent frame (following these same steps).
- If the name does not have a binding in the frame and the frame does not have a parent, a `SchemeNameError` is raised.

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

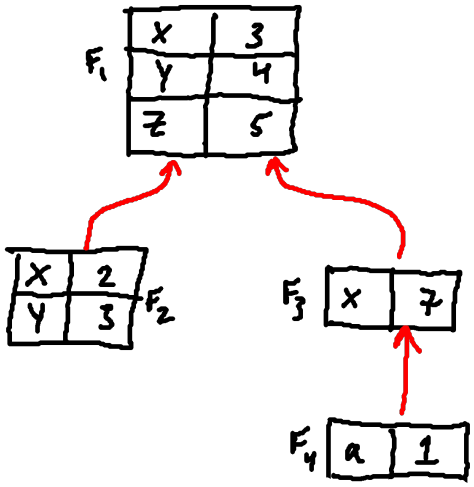
In order to make variables work properly, you will need to implement the kind of lookup described above in Python. To this end, **you should create a class to represent frames**.

> **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 frames 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 frame 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).

6.4.1) Frames: Example

The following shows an example of a frame structure, where arrows indicate each frame's parent, if any. Here we have four frames (each of which might be an instance of the class you created to represent frames), labeled **F1**, **F2**, **F3**, and **F4**. Both **F2** and **F3** have **F1** as a parent frame, and **F4** has **F3** as a parent frame. **F1** does not have a parent frame.



We say that the values `x`, `y`, and `z` are *bound* in **F1**. Note that `x` and `y` are bound in **F2**, `x` is bound in **F3**, and `a` is bound in **F4**. 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 **F4** gives us the value `1`, and looking up `z` in **F4** gives us `5`.

Notice that, as was mentioned above, the frame is crucial for determining the result of an expression.

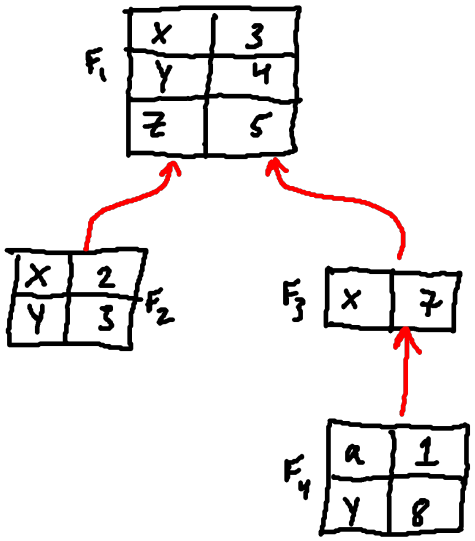
What is the result of evaluating `x` in **F1**?

What is the result of evaluating `x` in **F2**?

What is the result of evaluating `x` in **F3**?

What is the result of evaluating `x` in **F4**?

Also note that the `define` keyword always operates directly on the frame in which it is evaluated. If we were to evaluate `(define y 8)` in **F4**, this would result in a new binding inside of **F4** (without affecting the parent frames):



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

What is the result of evaluating `y` in **F1**?

What is the result of evaluating `y` in **F2**?

What is the result of evaluating `y` in **F3**?

What is the result of evaluating `y` in **F4**?

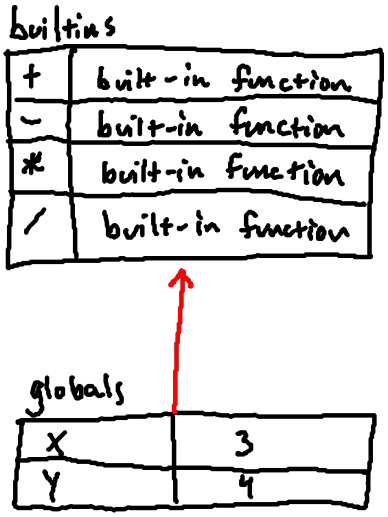
6.4.2) Frames: Initial Structure

Running a program in this language typically involves evaluating multiple expressions, one after the other, in the same frame, and so we will take a moment here to think about what the initial structure of these frames should look like.

When we first start a new REPL, we will need to think about two main frames: a frame to hold the *built-in* values (such as the `+` function) and a "global" frame where top-level definitions from users' programs will be bound. Then, every expression we provide to the REPL should be evaluated in that global frame.

For example, running the code below from the REPL should result in the frame 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 frame, we end up finding them in the built-ins frame.

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 a frame and modifying our interpreter so that it handles variables and the `define` keyword.

6.5.1) Changes to `evaluate`

Beyond implementing a representation for frames, 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 frame in which the expression should be evaluated. If no frame is passed in, an empty frame should be used, whose parent is a frame containing all the bindings from the `scheme_builtins` dictionary.
- make sure that `evaluate` handles the `define` keyword properly, evaluating the given expression and storing the result in the frame that was passed to `evaluate`.
- modify the way symbols are handled in `evaluate`, so that if the symbol exists as a key in the frame (or a parent frame), `evaluate` returns the associated value.

6.5.2) Examples

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

input: `(+ 3 2 4 (- 2 7 8))`
output:

input: `(define x (+ 2 3))`
output:

input: `(define x2 (* x x))`
output:

input: `(+ x2 x)`
output:

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 to make sure things are working as expected.

6.6) Frames, Test Cases, and the REPL

All of the tests from this point on involve using your interpreter to run small programs by evaluating multiple expressions one after the other in the same frame.

To make this kind of automatic checking possible, you should define a function called `make_initial_frame`. This function should take no arguments and should return a single new frame representing the global frame (i.e., an empty frame that has the builtins as its parent).

All test cases moving forward will use this function to simulate, in some sense, starting up a brand-new REPL and evaluating multiple expressions at that REPL in order. We do this by:

- Calling `make_initial_frame` to create a brand-new frame, and then
- calling `evaluate` for each subsequent expression, each time passing in the frame created in the first step (so that they all work in the same frame).

> **Real Programs in the Test Cases!**
>
> Note also that we are working with authentic programs here; and for each test case moving forward, there is a corresponding file in the `test_inputs` directory in the code distribution; these files contain the expressions that are evaluated in the associated test cases, 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. Note that you should be able to open these files in your normal text editor (and some editors may even provide syntax highlighting for Scheme!).
>
> If you are having trouble opening these files or figuring out why a given expression evaluates the way it does, please don't hesitate to ask for help!

6.6.1) Updating the REPL

Incorporating frames also requires changing our code for the REPL so that all of the inputs we provide are run in the same frame, so that variables we define at the REPL can be used in later expressions, like so:

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

in> (+ x 10)
    out> 17
```

As before, we have implemented this change for you; but it is necessary to change the value associated with `use_frames` on the last line of `lab.py` from `use_frames=False` to `use_frames=True`.

Also, if you want to be able to use the tab key to complete variable names (not just keywords), you should implement an `__iter__` method for your frame class that yields just the bound variable names (but not their values).

7) 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*.

Currently, the operations we can perform are limited to the functions in the `scheme_builtins` dictionary. We can really empower a user of the language by allowing them to define functions of their own. As we've seen throughout 6.101, the ability to define our own functions is a really powerful means of abstraction, and so this step will add a lot of power to our little language.

Note: we recommend reading all of sections 7.1-7.4 before implementing custom function definitions and calls.

7.1) Defining functions

In order to be able to make use of functions, we first need a way for a user to *define* a new function. We will accomplish this via the `lambda` special form.

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 expression 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 frame in which the function was defined (the function's enclosing frame)

You should define a class to represent your user-defined functions. Once again, it is up to you to determine exactly how to represent things inside of these functions; 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)[6].

> **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 frame 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 frame.

7.2) Calling Functions

Defining functions is nice, but we also need a way to *call* these functions after we've defined them. 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 frame (from which the function is being called).
- make a new frame whose parent is the function's enclosing frame (this is called lexical scoping).
- in that new frame, bind the function's parameters to the arguments that are passed to it.
- evaluate the body of the function in that new frame.

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, a `SchemeEvaluationError` 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. As such, you may find it helpful to review the readings from week 1 and week 2.

7.3) 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 frame?

--

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 frame in which that function was *created*, not the `7` from the frame 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?

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

--

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

--

Check Yourself:

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

Show/Hide

7.4) Changes to `evaluate`

We will also 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 frame 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 frame, or it should raise an `SchemeNameError` 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 frame, 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, a `SchemeEvaluationError` should be raised.

> **Note**
>
> 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 subsection 7.3. Once you are reasonably certain that everything is working, try them with `test.py`.

> **Check Yourself:**
>
> How difficult was it to add the `lambda` special form? And how much did it complicate your code? From here through Lisp Part 2 next week, we will be adding several new special forms to the language, so it is worth thinking ahead to see if there are ways of reorganizing your code to make adding new special forms easier.

7.5) 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`.

8) SchemeErrors

Note that throughout the lab we encourage you to raise various `SchemeErrors`. We are testing that:

- a `SchemeSyntaxError` is raised inside of parse to indicate that the provided tokens do not make a well-formed expression. Note that a well-formed expression consists of either a single symbol / number token or a single expression which begins with open parenthesis `'('` and finishes at the end of the tokens with a matching close `')'` parenthesis. If there are multiple expressions in the input tokens, even if they are well-formed, your code should raise a `SchemeSyntaxError`. For example, `parse(['(', 'expression1', ')', '(', 'expression2', ')'])` should raise a `SchemeSyntaxError` because the result of parsing the expression starting at index `0`, results in `['expression1']`, which ends at an index before the end of the tokens.

- a `SchemeNameError` is raised when in the course of evaluating an expression the program cannot find a binding for a given symbol (represented by a string). A `SchemeNameError` that arises in the course of a larger expression should over-ride a `SchemeEvaluationError`. For example `evaluate(['undefined_symbol', 3, 4])` should raise a `SchemeNameError`, not a `SchemeEvaluationError` because the interpreter should realize the binding for `'undefined_symbol'` does not exist before we attempt to call it as a function. Similarly, `evaluate(['+', 'undefined_symbol', 4])` should also raise a `SchemeNameError` because the program should realize the symbol is undefined before calling the add function.

- a `SchemeEvaluationError` is raised in `evaluate` when in the course of evaluating a nested expression (a list) we are unable to call a function due to the first part of the expression being missing or not being a callable function (e.g., `parse([])` or `parse([5, 4])`) or not providing the correct number of arguments for the given function (e.g., `evaluate(parse(tokenize('((lambda (x y) (* x y)) 1 2 3)')))` provides three arguments for the lambda function instead of the required two arguments.)

9) Endnotes

At this point, we have a very nice start toward an interpreter for Scheme. We have the ability to create 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 implementing the various pieces of the environment model we've been discussing throughout the semester!

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, along with a few other niceties.

10) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script. If you haven't already installed it, see the instructions on this page.

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:

```
$ 6.101-submit -a lisp_1 /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

> When this page was loaded, you had not yet made any submissions.
> If you make a submission, results should show up here automatically; or you may click here or reload the page to see updated results.

Footnotes

[1] The book associated with this subject, Structure and Interpretation of Computer Programs, is considered by many to be one of the best books ever written on the topic of programming.

[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 SchemeNameError("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). Note also that it is good style to avoid large try / except blocks.

[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: 1)Multiplication with 1 argument passed in should return that argument itself. 2)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: 1) Division with no arguments should raise an exception. 2) Division with a single argument x should return $1/x$.

[5] Note that this is equivalent to Python's "walrus operator," which was new in Python 3.8.

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