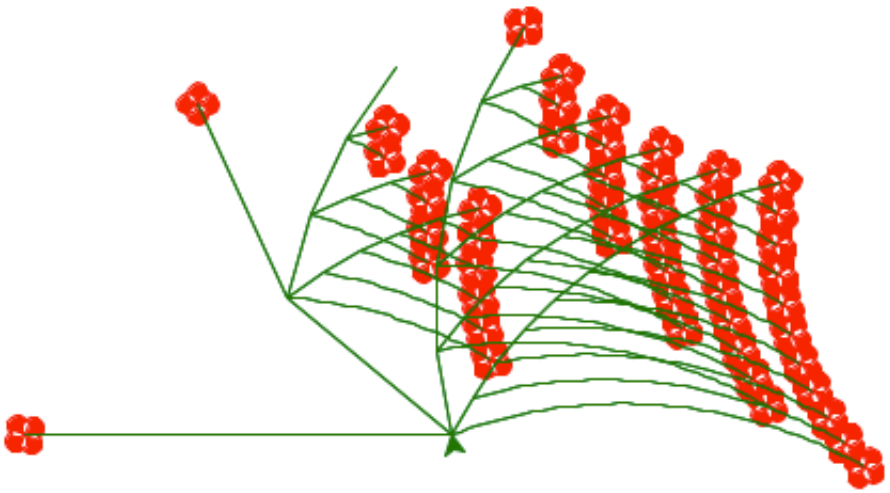
Project 4: The Scheme Interpreter

scheme.zip (scheme.zip)



Eval calls apply,
which just calls eval again!
When does it all end?

Introduction

Important submission note: For full credit:

- submit with Part I complete by **Thursday 11/9** (worth 1 pt),
- submit again with Part II complete by **Tuesday 11/14** (worth 1 pt), and
- submit the entire project by **Thursday 11/16**. You will get an extra credit point for submitting the entire project by Wednesday 11/15.

The Scheme project involves writing an interpreter for the Scheme language which is no small task! Start working on the project *now*! There are many parts and students often get stuck throughout the project so It's best to solve these problems early while there's still plenty of time. Remember that you can ask questions about the project in lab and office hours too!

We've also written a language specification (/~cs61a/fa17/articles/scheme-spec.html) and primitive procedure reference (/~cs61a/fa17/articles/scheme-primitives.html) for the CS 61A subset of Scheme that you'll be building in this project. Reading the entirety of either of these documents should not be necessary, but we'll point out useful sections from the documentation in each part of the project.

In this project, you will develop an interpreter for a subset of the Scheme language. As you proceed, think about the issues that arise in the design of a programming language; many quirks of languages are byproducts of implementation decisions in interpreters and compilers. The subset of the language used in this project is described in the functional programming (http://composingprograms.com/pages/32-functional-programming.html) section of Composing Programs. Since we only include a subset of the language, your interpreter will not exactly match the behavior of other interpreters.

You will also implement some small programs in Scheme. Scheme is a simple but powerful functional language. You should find that much of what you have learned about Python transfers cleanly to Scheme as well as to other programming languages.

The project concludes with an open-ended graphics contest that challenges you to produce recursive images in only a few lines of Scheme. As an example, the picture above abstractly depicts all the ways of making change for $0.50 using U.S. currency. All flowers appear at the end of a branch with length 50. Small angles in a branch indicate an additional coin, while large angles indicate a new currency denomination. In the contest, you too will have the chance to unleash your inner recursive artist.

This project includes several files, but all of your changes will be made to the first four: `scheme.py`, `scheme_reader.py`, `questions.scm`, and `tests.scm`. You can download all of the project code as a zip archive (scheme.zip), which contains the following files:

- `scheme.py`: the Scheme evaluator
- `scheme_reader.py`: the Scheme syntactic analyzer
- `questions.scm`: a collection of functions written in Scheme
- `tests.scm`: a collection of test cases written in Scheme
- `scheme_tokens.py`: a tokenizer for Scheme
- `scheme_primitives.py`: definitions for primitive Scheme procedures

- `buffer.py` : a Buffer implementation, used in `scheme_reader.py`
- `ucb.py` : utility functions for 61A
- `ok` : the autograder
- `tests` : a directory of tests used by `ok`
- `mytests.rst` : A space for you to add your custom tests in Python; see section on adding your own tests

In Parts I and II, you will develop the interpreter in several stages:

- Reading Scheme expressions
- Symbol evaluation
- Calling built-in procedures
- Definitions
- Lambda expressions and procedure definition
- Calling user-defined procedures
- Evaluation of special forms

In Part III, you will implement Scheme procedures.

Logistics

This is a 18-day project. You may work with one other partner. You should not share your code with students who are not your partner or copy from anyone else's solutions. In the end, you will submit one project for both partners.

Remember that you can earn an additional bonus point by submitting the project at least 24 hours before the deadline.

The project is worth 31 points. 29 points are assigned for correctness, and 2 points for the overall composition (../../articles/composition.html) of your program.

You will turn in the following files:

- `scheme_reader.py`
- `scheme.py`
- `questions.scm`
- `tests.scm`

You do not need to modify or turn in any other files to complete the project. To submit the project, run the following command:

```
python3 ok --submit
```

You will be able to view your submissions on the Ok dashboard (http://ok.cs61a.org).

For the functions that we ask you to complete, there may be some initial code that we provide. If you would rather not use that code, feel free to delete it and start from scratch. You may also add new function definitions as you see fit.

However, please do **not** modify any other functions. Doing so may result in your code failing our autograder tests. Also, please do not change any function signatures (names, argument order, or number of arguments).

Testing

Throughout this project, you should be testing the correctness of your code. It is good practice to test often, so that it is easy to isolate any problems. However, you should not be testing *too* often, to allow yourself time to think through problems.

We have provided an **autograder** called `ok` to help you with testing your code and tracking your progress. The first time you run the autograder, you will be asked to **log in with your Ok account using your web browser**. Please do so. Each time you run `ok`, it will back up your work and progress on our servers.

The primary purpose of `ok` is to test your implementations, but there are two things you should be aware of.

First, some of the test cases are *locked*. To unlock tests, run the following command from your terminal:

```
python3 ok -u
```

This command will start an interactive prompt that looks like:

```
=====================================================================
Assignment: The Scheme Interpreter
Ok, version ...
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Unlocking tests

At each "? ", type what you would expect the output to be.
Type exit() to quit

---------------------------------------------------------------------
Question 0 > Suite 1 > Case 1
(cases remaining: 1)

>>> Code here
?
```

At the `?`, you can type what you expect the output to be. If you are correct, then this test case will be available the next time you run the autograder.

The idea is to understand *conceptually* what your program should do first, before you start writing any code.

Once you have unlocked some tests and written some code, you can check the correctness of your program using the tests that you have unlocked:

```
python3 ok
```

Most of the time, you will want to focus on a particular question. Use the `-q` option as directed in the problems below.

We recommend that you submit **after you finish each problem**. Only your last submission will be graded. It is also useful for us to have more backups of your code in case you run into a submission issue.

The `tests` folder is used to store autograder tests, so **do not modify it**. You may lose all your unlocking progress if you do. If you need to get a fresh copy, you can download the zip archive (scheme.zip) and copy it over, but you will need to start unlocking from scratch.

If you do not want us to record a backup of your work or information about your progress, use the `--local` option when invoking `ok`. With this option, no information will be sent to our course servers.

Details of Scheme

Read-Eval-Print. The interpreter reads Scheme expressions, evaluates them, and displays the results.

```
scm> 2
2
scm> (+ 2 3)
5
scm> (((lambda (f) (lambda (x) (f f x)))
       (lambda (f k) (if (zero? k) 1 (* k (f f (- k 1)))))) 5)
120
```

The starter code for your Scheme interpreter in `scheme.py` can successfully evaluate the first expression above, since it consists of a single number. The second (a primitive call) and the third (a computation of 5 factorial) will not work just yet.

Load. Our `load` procedure differs from standard Scheme in that we use a symbol for the file name. For example, to load `tests.scm`, evaluate the following call expression.

```
scm> (load 'tests)
```

Symbols. Various dialects of Scheme are more or less permissive about identifiers (which serve as symbols and variable names).

Our rule is that:

> An identifier is a sequence of letters (a-z and A-Z), digits, and characters in `!$%&*/:<=>?@^_~-+.` that do not form a valid integer or floating-point numeral.

Our version of Scheme is case-insensitive: two identifiers are considered identical if they match except possibly in the capitalization of letters. They are internally represented and printed in lower case:

```
scm> 'Hello
hello
```

Turtle Graphics. In addition to standard Scheme procedures, we include procedure calls to the Python `turtle` package. This will come in handy for the contest.

You can read the turtle module documentation (http://docs.python.org/py3k/library/turtle.html) online.

Note: The `turtle` Python module may not be installed by default on your personal computer. However, the `turtle` module is installed on the instructional machines. So, if you wish to create turtle graphics for this project (i.e. for the contest), then you'll either need to setup `turtle` on your personal computer or use university computers.

Testing Your Scheme Interpreter

> To receive full composition credit, you are **required** to add and submit your own **Scheme** tests for the interpreter in `tests.scm`. This is different from the optional tests in `mytests.rst`.

Testing. The `tests.scm` file contains a long list of sample Scheme expressions and their expected values. Many of these examples are from Chapters 1 and 2 of Structure and Interpretation of Computer Programs (), the textbook that Composing Programs is adapted from.

```
(+ 1 2)
; expect 3
(/ 1 0)
; expect Error
```

You can compare the output of your interpreter to the expected output by running:

```
python3 ok -q tests.scm
```

For the example above, your Scheme interpreter will evaluate `(+ 1 2)` using your code in `scheme.py`, then output a test failure if `3` is not returned as the value. The second example tests for an error (but not the specific error message).

Only a small subset of tests are designated to run by default because `tests.scm` contains an `(exit)` call near the beginning, which halts testing. **As you complete more of the project, you should move or remove this call**. However, your interpreter doesn't know how to `exit` until Problems 3 and 4 are completed; all tests will run until then.

Writing Tests. As you proceed in the project, add new tests to the top of `tests.scm` to verify the behavior of your implementation. Half of your composition score for this project depends on whether you have tested your implementation in ways that are different from the `ok` tests and other Scheme assignments you may have seen in class.

Exceptions. As you develop your Scheme interpreter, you may find that Python raises various uncaught exceptions when evaluating Scheme expressions. As a result, your Scheme interpreter will halt. Some of these may be the results of bugs in your program, and some may be useful indications of errors in user programs. The former should be fixed (of course!) and the latter should be handled, usually by raising a `SchemeError`. All `SchemeError` exceptions are handled and printed as error messages by the `read_eval_print_loop` function in `scheme.py`. Ideally, there should *never* be unhandled Python exceptions for any input to your interpreter.

Your own test cases

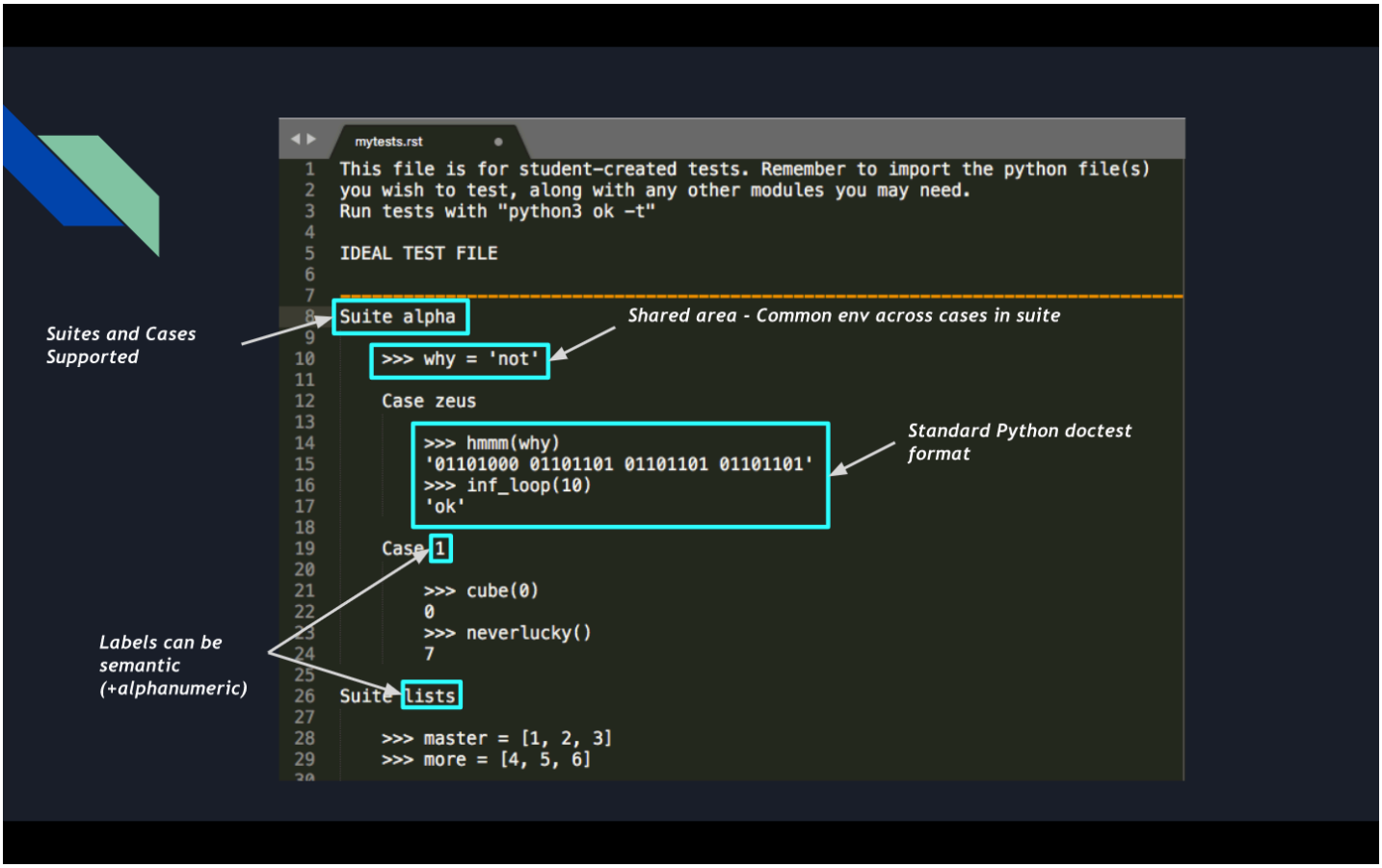
> Adding your own **Python** tests is entirely optional, and you will not submit your `mytests.rst` file. This is different from the **required** tests in `tests.scm`.

While the Scheme tests in `tests.scm` will test the *correctness* of your interpreter, they may not be of much use while you're still implementing basic features. Moreover, directly testing the implementation of the interpreter can be very useful in fixing some bugs.

We highly recommend that you add your own tests as you work through the project. It's a really helpful way to speed up the debugging process and improve your understanding of the code.

The course staff has also made an instructional video (https://youtu.be/o1TD0K1xWXs) summarizing the information below.

Adding tests



Adding tests is easy. Directly edit the `mytests.rst` file included with the Scheme project. We provide a sample structure to start from, but the test format is actually quite flexible. Here are some simple rules to follow:

- Follow standard Python doctest format. This is what we mostly use for the Ok tests, so feel free to use those as examples.
- For the most part, you may use whitespace as you'd like, but we recommend keeping it organized for your own sake.
- Generally, smaller, simpler tests will be more useful than larger, more complex tests.

You may also find our debugging guide (/~cs61a/fa17/articles/debugging.html#debugging-process) helpful. If you're stuck on a particularly tricky `Ok` test case, a good first step would be to break it up into small parts and test them out yourself in `mytests.rst`.

Running tests

To run all your tests in `mytests.rst` with verbose results:

```
python3 ok -t -v
```

If you put your tests in a different file or split your tests up into multiple files:

```
python3 ok -t your_new_filename.rst
```

To run just the tests from suite 1 case 1 in `mytests.rst`:

```
python3 ok -t --suite 1 --case 1
```

You might have noticed that there's a "test coverage" percentage for your tests (note that coverage statistics are only returned when running all tests). This is a measure of your test's code coverage (https://en.wikipedia.org/wiki/Code_coverage). If you're interested, you can find more information in our reference guide (/~cs61a/fa17/articles/using-ok.html#running-your-own-tests).

Running Your Scheme Interpreter

To start an interactive Scheme interpreter session, type:

```
python3 scheme.py
```

You can use your Scheme interpreter to evaluate the expressions in an input file by passing the file name as a command-line argument to `scheme.py`:

```
python3 scheme.py tests.scm
```

Currently, your Scheme interpreter can handle a few simple expressions, such as:

```
scm> 1
1
scm> 42
42
scm> true
True
```

To exit the Scheme interpreter, press `Ctrl-d` or evaluate the `exit` procedure (after completing problems 3 and 4):

```
scm> (exit)
```

Overview

Here is a brief overview of each of the Read-Eval-Print Loop components in our interpreter.

- **Read**: This step parses user input (a string of Scheme code) into our interpreter's internal Python representation of Scheme expressions (e.g. Pairs).
    - *Lexical analysis* has already been implemented for you in the `tokenize_lines` function in `scheme_tokens.py`. This function returns a `Buffer` (from `buffer.py`)

of tokens. You do not need to read or understand the code for this step.
    - *Syntactic analysis* happens in `scheme_reader.py`, in the `scheme_read` and `read_tail` functions. Together, these mutually recursive functions parse Scheme tokens into our interpreter's internal Python representation of Scheme expressions. You will complete both functions.
- **Eval**: This step evaluates Scheme expressions (represented in Python) to obtain values. Code for this step is in the main `scheme.py` file.
    - *Eval* happens in the `scheme_eval` function. If the expression being evaluated is a special form, the corresponding `do_XXX_form` function is called. You will fill in part of `scheme_eval`, as well as several of the `do_XXX_form` functions.
    - *Apply* happens in the `scheme_apply` function. `scheme_apply` calls the `apply` method of a `Procedure`, which is implemented in its subclasses `PrimitiveProcedure` and `UserDefinedProcedure`. For user-defined procedures, the `apply` method evaluates the procedure body, resulting in a mutually recursive eval-apply loop.
- **Print**: This step prints the `__str__` representation of the obtained value.
- **Loop**: The step is handled by the `read_eval_print_loop` function in `scheme.py`. You do not need to understand the entire implementation.

Part I: The Reader

> **Important submission note:** For full credit:
>
> - submit with Part I complete by **Thursday 11/9** (worth 1 pt),
> - submit again with Part II complete by **Tuesday 11/14** (worth 1 pt), and
> - submit the entire project by **Thursday 11/16**. You will get an extra credit point for submitting the entire project by Wednesday 11/15.
>
> The first part of this project deals with reading and parsing user input. All changes in this part should be made in `scheme_reader.py`.

`scheme_read` and `read_tail` in `scheme_reader.py` are mutually recursive functions. Together they parse Scheme code into Python values with the following representation:

Input Example	Scheme Data Type	Our Internal Representation
`scm> 1`	Numbers	Python's built-in `int` and `float` values
`scm> x`	Symbols	Python's built-in `string` values
`scm> true`	Booleans (`#t`, `#f`)	Python's built-in `True`, `False` values
`scm> (+ 2 3)`	Pairs	Instances of the `Pair` class, defined in `scheme_reader.py`

`scm> nil`	`nil`	The `nil` object, defined in `scheme_reader.py`

Both `scheme_read` and `read_tail` take in a single parameter, `src`, which is an instance of `Buffer` (see `buffer.py`). A `Buffer` is a *mutable* object, which keeps a record of all tokens that haven't been processed yet. There are two methods you'll use to interact with `src`:

- `src.remove_front()`: mutates `src` by removing the **first** token in `src` and returns it. For example, if `src` currently contains the tokens `[4, '.', 3, ')']`, then `src.remove_front()` will return `4`, and `src` will be left with `['.', 3, ')']`.
- `src.current()`: returns the **first** token in `src` without removing it. For example, if `src` currently contains the tokens `[4, '.', 3, ')']`, then `src.current()` will return `4` but `src` will remain the same.

Problem 1 (2 pt)

Implement the `scheme_read` and `read_tail` functions in `scheme_reader.py`.

The behavior of `scheme_read` depends on the type of the first token currently in `src`:

- If the token is the string `"nil"`, return the `nil` object.
- If the token is `(`, recursively call `read_tail` and return its result.
- If the next token is not a delimiter, then it must be self-evaluating. Return it. **(provided)**
- If none of the above cases apply, raise an error. **(provided)**

The behavior of `read_tail` also depends on the type of the first token currently in `src`:

- If there are no more tokens, raise an error. **(provided)**
- If the token is `)`, return the `nil` object.
- If the token is `.`, it is a dotted list. Implement this in Problem 2.
- If none of the above cases apply, the `src` is at the beginning of an expression. Then:
    1. Read the next expression (Hint: Which function do we use to read an expression?)
    2. Recursively read the rest of the original expression until the matching closing parenthesis.
    3. Return the results as a `Pair` instance.

Some implementation tips:

- `scheme_read` should be called when a complete Scheme expression needs to be extracted from `src`. It will remove enough tokens to form one expression and return that expression in the correct internal representation.
- `read_tail` expects to read the rest of a list or dotted list, assuming the open parenthesis of that list has already been removed by `scheme_read`. It will read expressions (and thus remove tokens) until the matching closing parenthesis `)` is seen. This list of expressions is returned in the correct internal representation (i.e. instances of the `Pair` class).

Test your understanding and implementation before moving on:

```
python3 ok -q 01 -u
python3 ok -q 01
```

Problem 2 (1 pt)

Complete the `read_tail` function by adding support for dotted lists.

- An ordinary list denotes a linked sequence of Pairs in which the `second` attribute of the final pair is `nil`.
- A dotted list denotes a sequence of Pairs in which the `second` attribute of the final pair may be any Scheme value.

For example:

> `(1 2 . 3)` should be converted to `Pair(1, Pair(2, 3))`

A dotted list must have exactly one item after the dot; anything else is a syntax error.

Consider the case of calling `scheme_read` on input "`(1 2 . 3)`". The `read_tail` function will be called on the suffix "`1 2 . 3)`", which is

- The pair consisting of the Scheme value `1` and the value of the tail "`2 . 3)`", which is
- The pair consisting of the Scheme value `2` and the Scheme value `3`.

Thus, `read_tail` would return `Pair(1, Pair(2, 3))`.

> *Hint*: In order to verify that only one element follows a dot, after encountering a `'.'`, read one additional expression and then check to see that a closing parenthesis follows.

Test your understanding and implementation before moving on:

```
python3 ok -q 02 -u
python3 ok -q 02
```

You should also test your parser by:

- Running the doctests for `scheme_reader.py`

  ```
  python3 -m doctest scheme_reader.py -v
  ```

- Testing interactively by running `python3 scheme_reader.py`. Every time you type in a value into the prompt, both the `str` and `repr` values of the parsed expression are printed. You can try the following inputs:

```
read> 42
str : 42
repr: 42
read> nil
str : ()
repr: nil
read> (1 (2 3) (4 (5)))
str : (1 (2 3) (4 (5)))
repr: Pair(1, Pair(Pair(2, Pair(3, nil)), Pair(Pair(4, Pair(Pair(5, nil), nil)),
read> (1 (9 8) . 7)
str : (1 (9 8) . 7)
repr: Pair(1, Pair(Pair(9, Pair(8, nil)), 7))
read> (hi there . (cs . (student)))
str : (hi there cs student)
repr: Pair('hi', Pair('there', Pair('cs', Pair('student', nil))))
```

One you have completed Part I, make sure you submit using OK to receive full credit for the first checkpoint.

```
python3 ok --submit
```

Part II: The Evaluator

> **Important submission note:** For full credit:
>
> - submit with Part I complete by **Thursday 11/9** (worth 1 pt),
> - submit again with Part II complete by **Tuesday 11/14** (worth 1 pt), and
> - submit the entire project by **Thursday 11/16**. You will get an extra credit point for submitting the entire project by Wednesday 11/15. All changes in this part should be made in `scheme.py`.

In the starter implementation given to you, the evaluator can only evaluate self-evaluating expressions: numbers, booleans, and `nil`.

Read the first two sections of `scheme.py`, called Eval/Apply and Environments.

- The `scheme_apply` function is complete, but part of `scheme_eval` and most of the functions or methods they use are not yet implemented.
- The `.apply` methods in subclasses of `Procedure` and the `make_call_frame` function assist in applying built-in and user-defined procedures.
- The `Frame` class implements an environment frame.
- The `LambdaProcedure` class (in the Procedures section) represents user-defined procedures.

These are all of the essential components of the interpreter; the rest of `scheme.py` defines special forms and input/output behavior.

Test your understanding of how these components fit together by unlocking the tests for `eval_apply`.

```
python3 ok -q eval_apply -u
```

Some Core Functionality

Problem 3 (1 pt)

Implement both `define` and `lookup` methods of the `Frame` class. Each `Frame` object has the following instance attributes:

- `bindings` is a dictionary that maps Scheme symbol keys (represented as Python strings) to Scheme values.
- `parent` is the parent `Frame` instance. The parent of the Global Frame is `None`.

`define` takes a symbol (represented by a Python string) and value and binds the value to that symbol in the frame.

`lookup` takes a symbol and returns the value bound to that name in the first `Frame` of the environment in which that name is found. Your implementation should:

- Return the value of the symbol in `self.bindings` if it exists.
- Otherwise, `lookup` that symbol in the `parent` if the `parent` exists.
- Otherwise, raise a `SchemeError`. **(provided)**

Test your understanding and implementation before moving on:

```
python3 ok -q 03 -u
python3 ok -q 03
```

After you complete this problem, you can open your Scheme interpreter (with `python3 scheme.py`). You should be able to look up built-in procedure names:

```
scm> +
#[+]
scm> odd?
#[odd?]
scm> display
#[display]
```

However, your Scheme interpreter will still not be able to apply these procedures. Let's fix that.

Problem 4 (1 pt)

Complete the `apply` method in the class `PrimitiveProcedure`, which is called by `scheme_apply`. Primitive procedures are applied by calling a corresponding Python function that implements the procedure. Instances of the `PrimitiveProcedure` class, defined in `scheme.py`, represent the values of Scheme primitive procedures. A `PrimitiveProcedure` has two instance attributes:

- `fn` is the *Python* function that implements the primitive Scheme procedure.
- `use_env` is a Boolean flag that indicates whether or not this primitive procedure will expect the current environment to be passed in as the last argument. The environment is required, for instance, to implement the primitive `eval` procedure.

To see a list of all Scheme primitive procedures used in the project, look in the `scheme_primitives.py` file. Any function decorated with `@primitive` will be added to the globally-defined `PRIMITIVES` list.

The `apply` method of `PrimitiveProcedure` takes a Scheme list of argument values, and the current environment. Your implementation should:

- Convert the Scheme list to a Python list of arguments. **(provided)**
- If `self.use_env` is `True`, then add the current environment `env` as the last argument to this Python list.
- Call `self.fn` on all of those arguments (*Hint*: Use `*args` notation).
- If calling the function results in a `TypeError` exception being raised, then the wrong number of parameters were passed. Intercept the exception and raise an appropriate `SchemeError` in its place.

Test your understanding and implementation before moving on:

```
python3 ok -q 04 -u
python3 ok -q 04
```

Problem 5 (1 pt)

`scheme_eval` evaluates a Scheme expression in a given environment. Most of `scheme_eval` has already been implemented for you. It currently looks up names in the current environment, returns self-evaluating expressions (like numbers) and evaluates special forms.

Implement the missing part of `scheme_eval`, which evaluates a call expression. To evaluate a call expression, we do the following:

1. Evaluate the operator (which should evaluate to a Scheme procedure)
2. Evaluate the operands
3. Apply the procedure on the evaluated operands.

The `check_procedure` function, which raises an error if the provided argument is not a Scheme procedure, may be useful to check that your operator is indeed a procedure. Once you have checked that it is, the rest of the operation (steps 2 and 3 above) should be carried out in the `eval_call` method of the `Procedure` class, which you must also complete.

The `map` method of `Pair` can apply a function to every item in a Scheme list. The `scheme_apply` function applies a Scheme procedure to some arguments.

Test your understanding and implementation before moving on:

```
python3 ok -q 05 -u
python3 ok -q 05
```

Your interpreter should now be able to evaluate primitive procedure calls, giving you the functionality of the Calculator language and more.

```
scm> (+ 1 2)
3
scm> (* 3 4 (- 5 2) 1)
36
scm> (odd? 31)
True
```

> Now would be a good time to start adding tests to `tests.scm`. For each new problem you complete from now on, add a few tests to the top of `tests.scm` to verify the behavior of your implementation.

Problem 6 (1 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#define) to understand the behavior of the `define` special form! This problem only provides the behavior for binding expressions, not procedures, to names.

There are two missing parts in the `do_define_form` function, which handles the `(define ...)` special forms. Implement **just the first part**, which binds names to values but does not create new procedures. `do_define_form` should return the name after performing the binding.

```
scm> (define tau (* 2 3.1415926))
tau
```

Test your understanding and implementation before moving on:

```
python3 ok -q 06 -u
python3 ok -q 06
```

You should now be able to give names to values and evaluate the resulting symbols. Note that `eval` takes a quoted expression and evaluates it (you can think of it as "removing" the quotes):

```
scm> (eval (define tau 6.28))
6.28
scm> (eval 'tau)
6.28
scm> tau
6.28
scm> (define x 15)
x
scm> (define y (* 2 x))
y
scm> y
30
scm> (+ y (* y 2) 1)
91
scm> (define x 20)
x
scm> x
20
```

Problem 7 (1 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#quote) to understand the behavior of the `quote` special form.

First, implement the `do_quote_form` function, which evaluates the `quote` special form. The `quote` special form returns its operand expression without evaluating it.

You should now be able to evaluate quoted expressions.

```
scm> (quote hello)
hello
scm> (quote (1 . 2))
(1 . 2)
scm> (quote (1 (2 three . (4 . 5))))
(1 (2 three 4 . 5))
scm> (car (quote (a b)))
a
```

Next, complete your implementation of `scheme_read` in `scheme_reader.py` by handling one last case:

- If the token is a single quote (`'`), such as the first character of `'bagel` , then return

a `Pair` that wraps the Scheme expression after the quote (which you can get by recursively calling `scheme_read`) in `quote` (e.g. `Pair('quote', Pair('bagel', nil))`.

After completing your `scheme_read` implementation, the following quoted expressions should now work as well.

```
scm> 'hello
hello
scm> '(1 . 2)
(1 . 2)
scm> '(1 (2 three . (4 . 5)))
(1 (2 three 4 . 5))
scm> (car '(a b))
a
scm> (eval (cons 'car '('(1 2))))
1
```

Test your understanding and implementation before moving on:

```
python3 ok -q 07 -u
python3 ok -q 07
```

At this point in the project, your Scheme interpreter should support the following features:

- Evaluate atoms, which include numbers, booleans, nil, and symbols,
- Evaluate the `quote` special form,
- Define symbols, and
- Call primitive procedures, for example evaluating `(+ (- 4 2) 5)`.

User-Defined Procedures

User-defined procedures are represented as instances of the `LambdaProcedure` class. A `LambdaProcedure` instance has three instance attributes:

- `formals` is a Scheme list of the formal parameters (symbols) that name the arguments of the procedure.
- `body` is a Scheme list of expressions; the body of the procedure.
- `env` is the environment in which the procedure was **defined**.

Problem 8 (1 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#begin) to understand the behavior of the `begin` special form!

Implement the `eval_all` function (which is called from `do_begin_form`) to complete the implementation of the `begin` special form. A `begin` expression is evaluated by evaluating all sub-expressions in order. The value of the `begin` expression is the value of the final sub-expression.

```
scm> (begin (+ 2 3) (+ 5 6))
11
scm> (define x (begin (display 3) (newline) (+ 2 3)))
3
x
scm> (+ x 3)
8
scm> (begin (print 3) '(+ 2 3))
3
(+ 2 3)
```

If `eval_all` is passed an empty list of expressions (`nil`), then it should return the Python value `None`, which represents an undefined Scheme value.

Test your understanding and implementation before moving on:

```
python3 ok -q 08 -u
python3 ok -q 08
```

Problem 9 (1 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#lambda) to understand the behavior of the `lambda` special form!

Implement the `do_lambda_form` function, which creates `LambdaProcedure` instances. While you cannot call a user-defined procedure yet, you can verify that you have created the procedure correctly by typing a lambda expression into the interpreter prompt:

```
scm> (lambda (x y) (+ x y))
(lambda (x y) (+ x y))
```

In Scheme, it is legal to place more than one expression in the body of a procedure (there must be at least one expression). The `body` attribute of a `LambdaProcedure` instance is a Scheme list of body expressions.

Test your understanding and implementation before moving on:

```
python3 ok -q 09 -u
python3 ok -q 09
```

Problem 10 (2 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#define) to understand the behavior of the `define` special form! In this problem, we'll finish defining the `define` form for procedures.

Currently, your Scheme interpreter is able to bind symbols to user-defined procedures in the following manner:

```
scm> (define f (lambda (x) (* x 2)))
f
```

However, we'd like to be able to use the shorthand form of defining named procedures:

```
scm> (define (f x) (* x 2))
f
```

Modify the `do_define_form` function so that it correctly handles the shorthand procedure definition form above. Make sure that it can handle multi-expression bodies.

Test your understanding and implementation before moving on:

```
python3 ok -q 10 -u
python3 ok -q 10
```

You should now find that defined procedures evaluate to `LambdaProcedure` instances.

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

Problem 11 (2 pt)

Implement the `make_child_frame` method of the `Frame` class, which:

- Creates a new `Frame` instance, the parent of which is `self`. **(provided)**
- If the number of argument values does not match with the number of formal parameters, raises a `SchemeError`.
- Binds formal parameters to their corresponding argument values in the newly created frame.

Test your understanding and implementation before moving on:

```
python3 ok -q 11 -u
python3 ok -q 11
```

Problem 12 (1 pt)

Implement the `make_call_frame` method in `LambdaProcedure`, which is needed by the `apply` method defined in `UserDefinedProcedure`. It should create a new `Frame` instance using the `make_child_frame` method of the appropriate parent frame, binding formal parameters to argument values.

Since lambdas are lexically scoped, your new frame should be a child of the frame in which the lambda is defined. The `env` provided as an argument to `make_call_frame` is instead the frame in which the procedure is called, which will be useful when you implement a dynamically scoped procedure in problem 16.

Test your understanding and implementation before moving on:

```
python3 ok -q 12 -u
python3 ok -q 12
```

At this point in the project, your Scheme interpreter should support the following features:

- Create procedures using `lambda` expressions,
- Define named procedures using `define` expressions, and
- Call user-defined procedures.

> Now is an excellent time to revisit the tests in `tests.scm` and ensure that you pass the tests that involve definition (Sections 1.1.2 and 1.1.4). You should also add **additional tests of your own at the top of `tests.scm`** to verify that your interpreter is behaving as you expect. This is **required to earn full composition credit.**
>
> To run your tests, run the command:
>
> ```
> python3 ok -q tests.scm
> ```

Special Forms

Logical special forms include `if`, `and`, `or`, and `cond`. These expressions are special because not all of their sub-expressions may be evaluated.

In Scheme, only `False` is a false value. All other values (including `0` and `nil`) are true values. You can test whether a value is a true or false value using the provided Python functions `scheme_truep` and `scheme_falsep`, defined in `scheme_primitives.py`.

> Note: Scheme traditionally uses `#f` to indicate the false Boolean value. In our interpreter, that is equivalent to `false` or `False`. Similarly, `true`, `True`, and `#t` are all equivalent.

To get you started, we've provided an implementation of the `if` special form in the `do_if_form` function. Make sure you understand that implementation before starting the following questions.

Problem 13 (2 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#and) to understand the behavior of the `and` and `or` special forms!

Implement `do_and_form` and `do_or_form` so that `and` and `or` expressions are evaluated correctly.

The logical forms `and` and `or` are *short-circuiting*. For `and`, your interpreter should evaluate each sub-expression from left to right, and if any of these evaluates to a false value, then `False` is returned. Otherwise, it should return the value of the last sub-expression. If there are no sub-expressions in an `and` expression, it evaluates to `True`.

```
scm> (and)
True
scm> (and 4 5 6)  ; all operands are true values
6
scm> (and 4 5 (+ 3 3))
6
scm> (and True False 42 (/ 1 0))  ; short-circuiting behavior of and
False
```

For `or`, evaluate each sub-expression from left to right. If any sub-expression evaluates to a true value, return that value. Otherwise, return `False`. If there are no sub-expressions in an `or` expression, it evaluates to `False`.

```
scm> (or)
False
scm> (or 5 2 1)  ; 5 is a true value
5
scm> (or False (- 1 1) 1)  ; 0 is a true value in Scheme
0
scm> (or 4 True (/ 1 0))  ; short-circuiting behavior of or
4
```

Test your understanding and implementation before moving on:

```
python3 ok -q 13 -u
python3 ok -q 13
```

Problem 14 (2 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#cond) to understand the behavior of the `cond` special form!

Fill in the missing parts of `do_cond_form` so that it returns the value of the first result sub-expression corresponding to a true predicate, or the result sub-expression corresponding to `else`. Some special cases:

- When the true predicate does not have a corresponding result sub-expression, return the predicate value.
- When a result sub-expression of a `cond` case has multiple expressions, evaluate them all and return the value of the last expression. (*Hint*: Use `eval_all`.)

Your implementation should match the following examples and the additional tests in `tests.scm`.

```
scm> (cond ((= 4 3) 'nope)
           ((= 4 4) 'hi)
           (else 'wait))
hi
scm> (cond ((= 4 3) 'wat)
           ((= 4 4))
           (else 'hm))
True
scm> (cond ((= 4 4) 'here (+ 40 2))
           (else 'wat 0))
42
```

The value of a `cond` is undefined if there are no true predicates and no `else`. In such a case, `do_cond_form` should return `None`.

```
scm> (cond (False 1) (False 2))
scm>
```

Test your understanding and implementation before moving on:

```
python3 ok -q 14 -u
python3 ok -q 14
```

Problem 15 (2 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#let) to understand the behavior of the `let` special form!

The `let` special form binds symbols to values locally, giving them their initial values. For example:

```
scm> (define x 5)
x
scm> (define y 'bye)
y
scm> (let ((x 42)
           (y (* x 10)))  ; x refers to the global value of x, not 42
       (list x y))
(42 50)
scm> (list x y)
(5 bye)
```

Implement `make_let_frame`, which returns a child frame of `env` that binds the symbol in each element of `bindings` to the value of its corresponding expression. The `bindings` scheme list contains pairs that each contain a symbol and a corresponding expression.

You may find the following functions and methods useful:

- `check_form`: this function can be used to check the structure of each binding.
- `check_formals`: this function checks that formal parameters are a Scheme list of symbols for which each symbol is distinct.
- `make_child_frame`: this method (which you implemented in Problem 11) takes a `Pair` of formal parameters (symbols) and a `Pair` of values, and returns a new frame with all the symbols bound to the corresponding values.

Test your understanding and implementation before moving on:

```
python3 ok -q 15 -u
python3 ok -q 15
```

Problem 16 (1 pt)

> Read the Scheme Specifications (/~cs61a/fa17/articles/scheme-spec.html#mu) to understand the behavior of the `mu` special form!

Implement `do_mu_form` to evaluate the `mu` special form, a non-standard Scheme expression type. A `mu` expression is similar to a `lambda` expression, but evaluates to a `MuProcedure` instance that is **dynamically scoped**. Most of the `MuProcedure` class has been provided for you.

Complete the `MuProcedure` class so that when a call on such a procedure is executed, it is dynamically scoped. Calling a `LambdaProcedure` uses lexical scoping: the parent of the new call frame is the environment in which the procedure was **defined**. Calling a `MuProcedure` created by a `mu` expression uses dynamic scoping: the parent of the new call frame is the environment in which the call expression was **evaluated**. As a result, a `MuProcedure` does not need to store an environment as an instance attribute. It can refer to names in the environment from which it was called.

```
scm> (define f (mu (x) (+ x y)))
f
scm> (define g (lambda (x y) (f (+ x x))))
g
scm> (g 3 7)
13
```

Looking at `LambdaProcedure` should give you a clue about what needs to be done to `MuProcedure` to complete it. Test your understanding and implementation before moving on:

```
python3 ok -q 16 -u
python3 ok -q 16
```

Congratulations! Your Scheme interpreter implementation is now complete!

> The autograder tests for the interpreter are *not* comprehensive, so you may have uncaught bugs in your implementation. You should have been adding tests to the top of `tests.scm` as you did each problem, which will help you discover bugs on your own. **The tests that you have written tests will be evaluated as part of your composition score for the project.**
>
> To run your tests, run the command:
>
> ```
> python3 ok -q tests.scm
> ```
>
> Make sure to remove all of the `(exit)` commands, so that all the tests are run! We've provided 115 tests (not counting the extra credit tests), so if you don't see at least that many tests passed, you haven't removed all the `(exit)` commands. (Of course, you should have many more than that, since you've been writing your own as well.)

One you have completed Part II, make sure you submit using OK to receive full credit for the second checkpoint.

```
python3 ok --submit
```

Part III: Write Some Scheme

> Not only is your Scheme interpreter itself a tree-recursive program, but it is flexible enough to evaluate *other* recursive programs. Implement the following procedures in Scheme in the `questions.scm` file.
>
> In addition, for this part of the project, you may find the primitive procedure reference () very helpful if you ever have a question about the behavior of a built-in Scheme procedure, like the difference between `pair?` and `list?`.

The autograder tests for the interpreter are *not* comprehensive, so you may have uncaught bugs in your implementation. Therefore, you may find it useful to test your code for these questions in the staff interpreter or the web interpreter (http://scheme.cs61a.org/editor.html) and then try it in your own interpreter once you are confident your Scheme code is working.

Problem 17 (1 pt)

Implement the `enumerate` procedure, which takes in a list of values and returns a list of two-element lists, where the first element is the index of the value, and the second element is the value itself.

```
scm> (enumerate '(3 4 5 6))
((0 3) (1 4) (2 5) (3 6))
scm> (enumerate '())
()
```

Test your implementation before moving on:

```
python3 ok -q 17
```

Problem 18 (2 pt)

Implement the `list-change` procedure, which lists all of the ways to make change for a positive integer `total` amount of money, using a list of currency denominations, which is sorted in descending order. The resulting list of ways of making change should also be returned in descending order.

To make change for 10 with the denominations (25, 10, 5, 1), we get the possibliites:

```
10
5, 5
5, 1, 1, 1, 1, 1
1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

To make change for 5 with the denominations (4, 3, 2, 1), we get the possibilities:

```
4, 1
3, 2
3, 1, 1
2, 2, 1
2, 1, 1, 1
1, 1, 1, 1, 1
```

You may find that implementing a helper function, `cons-all`, will be useful for this problem. To implement `cons-all`, use the built-in map procedure (/~cs61a/fa17/articles/scheme-primitives.html#map). `cons-all` takes in an element `first` and a list of lists `rests`, and adds `first` to the beginning of each list in `rests`:

```
scm> (cons-all 1 '((2 3) (2 4) (3 5)))
((1 2 3) (1 2 4) (1 3 5))
```

You may also find the built-in append procedure (/~cs61a/fa17/articles/scheme-primitives.html#append) useful.

Test your implementation before moving on:

```
python3 ok -q 18
```

Problem 19 (2 pt)

In Scheme, source code is data. Every non-primitive expression is a list, and we can write procedures that manipulate other programs just as we write procedures that manipulate lists.

Rewriting programs can be useful: we can write an interpreter that only handles a small core of the language, and then write a procedure that converts other special forms into the core language before a program is passed to the interpreter.

For example, the `let` special form is equivalent to a call expression that begins with a `lambda` expression. Both create a new frame extending the current environment and evaluate a body within that new environment.

```
(let ((x 42) (y 16)) (+ x y))
;; Is equivalent to:
((lambda (x y) (+ x y)) 42 16)
```

We can use this rule to rewrite all `let` special forms into `lambda` expressions. We prevent evaluation of a program by quoting it, and then pass it to a procedure called `let-to-lambda`:

```
scm> (let-to-lambda '(let ((a 1) (b 2)) (+ a b)))
((lambda (a b) (+ a b)) 1 2)
scm> (let-to-lambda '(let ((a 1)) (let ((b a)) b)))
((lambda (a) ((lambda (b) b) a)) 1)
```

In order to handle all programs, `let-to-lambda` must be aware of Scheme syntax. Since Scheme expressions are recursively nested, `let-to-lambda` must also be recursive. In fact, the structure of `let-to-lambda` is somewhat similar to that of `scheme_eval` --but in Scheme!

```
(define (let-to-lambda expr)
  (cond ((atom?   expr) <rewrite atoms>)
        ((quoted? expr) <rewrite quoted expressions>)
        ((lambda? expr) <rewrite lambda expressions>)
        ((define? expr) <rewrite define expressions>)
        ((let?    expr) <rewrite let expressions>)
        (else           <rewrite other expressions>)))
```

Implement the `let-to-lambda` procedure, which takes in an expression and rewrites all of the `let` special forms in the expression into their equivalent `lambda` expressions. As a reminder, atoms include numbers, booleans, nil, and symbols.

Hint: You may want to implement `zip` at the top of `questions.scm` and also use the built-in `map` procedure.

```
scm> (zip '((1 2) (3 4) (5 6)))
((1 3 5) (2 4 6))
scm> (zip '((1 2)))
((1) (2))
scm> (zip '())
(() ())
```

Test your understanding and implementation before moving on:

```
python3 ok -q 19 -u
python3 ok -q 19
```

Note: We used `let` while defining `let-to-lambda`. What if we want to run `let-to-lambda` on an interpreter that does not recognize `let`? We can pass `let-to-lambda` to itself to rewrite itself into an *equivalent program without* `let`:

```
;; The let-to-lambda procedure
(define (let-to-lambda expr)
  ...)

;; A list representing the let-to-lambda procedure
(define let-to-lambda-code
  '(define (let-to-lambda expr)
     ...))

;; An let-to-lambda procedure that does not use 'let'!
(define let-to-lambda-without-let
  (let-to-lambda let-to-lambda-code))
```

Part IV: Extra Credit

Problem 20 (2 pt)

Complete the function `scheme_optimized_eval` in `scheme.py`. This alternative to `scheme_eval` is properly tail recursive. That is, the interpreter will allow an unbounded number of active tail calls (http://en.wikipedia.org/wiki/Tail_call) in constant space.

The `Thunk` class represents a thunk (http://en.wikipedia.org/wiki/Thunk), an expression that needs to be evaluated in an environment. When `scheme_optimized_eval` receives an expression in a `tail` context, then it returns an `Thunk` instance. Otherwise, it repeatedly evaluates expressions within the body of a `while` statement, updating `result` in each iteration.

A successful implementation will require changes to several other functions, including some functions that we provided for you. All tail calls should call `scheme_eval` with `True` as a third argument, indicating a tail call. Think about what Scheme expressions are in a tail context!

Once you finish, uncomment the following line in `scheme.py` to use your implementation:

```
scheme_eval = scheme_optimized_eval
```

Test your understanding and implementation before moving on:

```
python3 ok -q 20 -u
python3 ok -q 20
```

Problem 21 (1 pt)

Macros allow the language itself to be extended by the user. Simple macros can be provided with the `define-macro` special form. This must be used like a function definition, and it creates a procedure just like `define`. However, this procedure has a special evaluation rule: it is applied to its arguments without first evaluating them. Then the result of this application is evaluated.

This final evaluation step takes place in the caller's frame, as if the return value from the macro was literally pasted into the code in place of the macro.

Here is a simple example:

```
scm> (define (map f lst) (if (null? lst) nil (cons (f (car lst)) (map f (cdr lst)))))
scm> (define-macro (for formal iterable body)
....     (list 'map (list 'lambda (list formal) body) iterable))
scm> (for i '(1 2 3)
....     (print (* i i)))
1
4
9
(None None None)
```

The code above defines a macro `for` that acts as a `map` except that it doesn't need a lambda around the body.

In order to implement `define-macro`, implement the `MacroProcedure` class by overriding the `eval_call` method so that it applies the procedure directly to the arguments without evaluating them first. It should then evaluate and return the result.

> *Hint*: If you implemented Tail Recursion, you should use `complete_eval`!

Then, complete the implementation for `do_define_macro`, which should create a `MacroProcedure` and bind it to the given name as in `do_define_form`.

Test your understanding and implementation before moving on:

```
python3 ok -q 21 -u
python3 ok -q 21
```

Conclusion

Congratulations! You have just implemented an interpreter for an entire language!

Extra Challenge

We've implemented a significant subset of Scheme in this project, but your interpreter can be extended with even more features! If you enjoyed this project, we have some suggestions in the extension instructions (extensions.html).

Staff (/~cs61a/fa17/staff.html)

Resources (/~cs61a/fa17/resources.html)

Studying Guide (/~cs61a/fa17/articles/studying.html)

Debugging Guide (/~cs61a/fa17/articles/debugging.html)

Composition Guide (/~cs61a/fa17/articles/composition.html)

Policies (/~cs61a/fa17/articles/about.html)

Assignments (/~cs61a/fa17/articles/about.html#assignments)

Exams (/~cs61a/fa17/articles/about.html#exams)

Grading (/~cs61a/fa17/articles/about.html#grading)