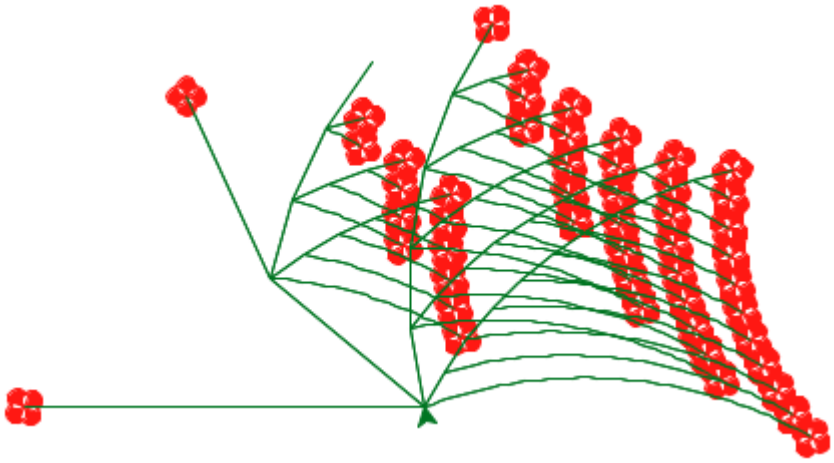
Project 4: Scheme Interpreter (Challenge Version) **scheme_stubbed.zip (scheme_stubbed.zip)**



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

Introduction

Important submission note: For full credit:

- submit with Part I done (including passing all tests provided in `tests.scm`) by **Friday, 11/19** (worth 2 pts), and
- submit the entire project by **Tuesday, 11/23**. You will get an extra credit point for submitting the entire project by Monday, 11/22.

Unlike the standard version of the project, there is only one checkpoint, not two.

We've written a language specification (/articles/scheme-spec/) and built-in procedure reference (/articles/scheme-builtins/) for the CS 61A subset of Scheme that you'll be building in this project. You will not be responsible for implementing everything in these documents, but what you do implement should be consistent with the descriptions here.

This is an alternate "extreme" version of the standard Scheme project that gives you *much* less guidance than the normal version. Traditionally, students without substantial prior programming experience have found this version of the project very difficult. Completing this version is, for grading purposes, equivalent to completing the standard version of Project 4. Completing this version will not give you any more credit than is possible by completing the standard version - it's just here if you want a challenging experience.

Part I will contain very little provided code. Part II, writing programs in Scheme, will be identical to the standard version.

You should not expect much assistance from staff if you choose to complete this version of the project. You can always switch to the standard version if you get stuck.

As a disclaimer, this version has not been tested to the same extent as the main project. If you believe you've found an error in the specifications, tests, or provided files, please let us know on Piazza and we will get it fixed as soon as possible.

> When students in the past have tried to implement the functions without thoroughly reading the problem description, they've often run into issues. 😱 **Read each description thoroughly before starting to code.**

Download starter files

You can download all of the project code as a zip archive (scheme_stubbed.zip).

Files you will edit:

- `scheme_eval_apply.py` : the recursive evaluator for Scheme expressions
- `scheme_forms.py` : evaluation for special forms
- `scheme_classes.py` : classes that describe Scheme expressions
- `questions.scm` : contains skeleton code for Part II

The rest of the files in the project:

- `scheme.py` : the interpreter REPL
- `pair.py` : defines the `Pair` class and the `nil` object
- `scheme_builtins.py` : built-in Scheme procedures
- `scheme_reader.py` : the reader for Scheme input (this file is obfuscated so that you can implement it in lab)
- `scheme_tokens.py` : the tokenizer for Scheme input
- `scheme_utils.py` : functions for inspecting Scheme expressions
- `ucb.py` : utility functions for use in 61A projects
- `tests.scm` : a collection of test cases written in Scheme
- `ok` : the autograder
- `tests` : a directory of tests used by `ok`
- `mytests.rst` : a file where you can add your own tests

Logistics

The project is worth 30 points. 28 points are assigned for correctness, which is including 1 point for passing `tests.scm`. 2 points are assigned for submitting Part I by the checkpoint.

Additionally, there are some extra credit point opportunities. You can get 1 EC point for submitting the entire project by **Monday, November 22**, and 2 EC points for submitting the extra credit problem.

> **Important:** In order to receive all of the extra credit points for Scheme, your implementation of the entire project, including the EC problem, must be submitted by the early submission deadline.

You will turn in the following files:

- `scheme_eval_apply.py`
- `scheme_forms.py`
- `scheme_classes.py`
- `questions.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).

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.

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. **If you forget to submit, your last backup will be automatically converted to a submission.**

If you do not want us to record a backup of your work or information about your progress, you can run

```
python3 ok --local
```

With this option, no information will be sent to our course servers. If you want to test your code interactively, you can run

```
python3 ok -q [question number] -i
```

with the appropriate question number (e.g. `01`) inserted. This will run the tests for that question until the first one you failed, then give you a chance to test the functions you wrote interactively.

You can also use the debugging print feature in OK by writing

```
print("DEBUG:", x)
```

which will produce an output in your terminal without causing OK tests to fail with extra output.

Interpreter details

Scheme features

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

```
scm> 2
2
scm> (+ 2 3)
5
scm> ((lambda (x) (* x x)) 5)
25
```

The starter code for your Scheme interpreter can successfully evaluate the first expression above, since it consists of a single number. The second (a call to a built-in procedure) and the third (a computation of 5 squared) will not work just yet.

Load. You can load a file by passing in 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 and are not existing special form shorthands.

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 **do not** have to install this package in order to participate.

If you're curious, you can read the turtle module documentation (http://docs.python.org/py3k/library/turtle.html) online.

Running the interpreter

To start an interactive Scheme interpreter session, type:

```
python3 scheme.py
```

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

```
scm> 1
1
scm> 42
42
scm> true
#t
```

To exit the Scheme interpreter, press `Ctrl-d` or evaluate the `exit` procedure:

```
scm> (exit)
```

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
```

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 (https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book-Z-H-4.html#%_toc_start), the textbook from which Composing Programs is adapted.

Part I: The Evaluator

In `scheme_eval_apply.py` we've provided a function definition for `scheme_eval` - you should not change the signature of this function, as it is called in the read-eval-print-loop. However, the implementation of this function is up to you. It should be able to evaluate atomic expressions and combinations, including self-evaluating expressions, names, call expressions, and special forms.

Problem 1 (8 pt)

In this problem, you will implement the core functionality of the interpreter. You should fill in the `scheme_eval` function and add any necessary functions/classes so that your interpreter is able to do the following:

- Evaluate self-evaluating atomic expressions including numbers, booleans, and nil
- Evaluate symbols by looking up their value in the current environment
- Apply built-in procedures (e.g. `+` and `cons`)
- Evaluate call expressions
- Add bindings to the current environment using the `define` special form.
- Evaluate the `quote` special form

At this point, you do not need to worry about creating user-defined procedures using the `define` special form (although you will in the next part). That is, your interpreter should be able to handle expressions such as `(define x 1)` but not `(define (foo x) 1)` after this question.

Remember to refer to the Scheme Specifications (/articles/scheme-spec/#define) in order to determine the behavior of `define` (and other special forms).

We've provided a few classes that you will use in this part:

- The `Frame` class is used to contain and organize the bindings in a specific frame. An instance of the `Frame` class is passed in to `scheme_eval` as `env`.
- The `BuiltinProcedure` class inherits from the `Procedure` class (since your interpreter should handle both user-defined and built-in procedures). The constructor for a `BuiltinProcedure` creates two instance attributes
  - `py_func` is a *Python* function implementing the built-in scheme procedure
  - `expect_env` is a Boolean that indicates whether or not the built-in procedure expects the current environemnt to be passed in as the last argument. The environment is required, for instance, to implement the built-in `eval` procedure.

You may add any attributes or methods to these classes you see fit in order to implement the above functionality.

Here are some other tips for this question:

- To see a list of all Scheme built-in procedures used in the project, look in the `scheme_builtins.py` file. Any function decorated with `@builtin` will be added to the globally-defined `BUILTINS` list. You can use any of these procedures in your tests.
- You may want to take a look at some methods contained in the `Pair` class - for example, the `map` method of `Pair` can apply a *one-argument function* to every item in a Scheme list.
- While built-in procedures follow the normal rules of evaluation (evaluate operator, evaluate operands, apply operator to operands), applying the operator does *not* create a new frame.
- In order to implement the `quote` special form, you will have to both evaluate the expression correctly in the `scheme_eval` function and make sure your parser is able to correctly form these expressions. The output of the parser should substitute an equivalent expression using the `quote` keyword if it sees the following token: `'`.
- How you implement special forms is up to you, but we recommend you encapsulate the logic for each special form separately somehow.

Use Ok to test your code:

```
python3 ok -q 01
```

After you complete this problem, your interpreter should be able to evalate the following expressions:

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

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

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

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

Problem 2 (7 pt)

In this problem, you will implement user-defined expressions and some related features. After this, your interpreter should be able to accomplish the following:

- Evaluating `begin` and `lambda` special forms
- Creating user-defined functions when evaluating the `define` special form
- Applying lambda functions and user-defined procedures to arguments in a call expression

Although you added some functionality for call expressions in the previous part, user-defined procedures require some special handling. In particular, built-in procedures do *not* require creating new frames when you call them. However, user-defined procedures will require creating a new `Frame` (which we will use in accordance with the rules for calling functions we've learned in the class so far).

Here are some additional hints and clarifications:

- A `begin` special form should evaluate to an undefined value if there are no sub-expressions to evaluate. The way we will represent this in the interpreter is by returning the Python value `None`.
- User-defined procedures in Scheme are the same as lambda procedures. For example, the expression `(define (foo x) x)` binds the value `(lambda (x) x)` to the name `foo` in the current environment.
- The body of a procedure can contain multiple expressions, which will be represented as a list of expressions. Only the value that the final expression evaluates to will be returned by the function call.

Here are some examples of expressions your interpreter should now be able to evaluate:

```
scm> (begin (print 3) '(+ 2 3))
3
(+ 2 3)
scm> (define x (begin (display 3) (newline) (+ 2 3)))
3
x

scm> (lambda (x y) (+ x y))
(lambda (x y) (+ x y))
scm> ((lambda (x y) (+ x y)) 1 2)
3

scm> (define (square x) (* x x))
square
scm> square
(lambda (x) (* x x))
scm> (square 4)
16

scm> (define (print-twice x) (print x) (print x))
print-twice
scm> (print-twice 1)
1
1
```

Use Ok to test your code:

```
python3 ok -q 02
```

Problem 3 (8 pt)

In this part, you will be implementing the following special forms:

- `if`
- `and` / `or`
- `if` / `cond`
- `let`
- `mu`

Make sure to read the Scheme Specifications (/articles/scheme-spec/) for information on these special forms. Here are some clarifications on their behavior which are not mentioned in the specifications.

- `and` and `or` should exhibit short-circuiting behavior as in Python and evaluate left to right.
- `mu` procedures follow the same evaluation rules as lambda procedures (evaluate the operator, evaluate the operand, apply the operator to the operands). However, `mu` procedures are *dynamically scoped* - meaning the `Frame` created by calling a `mu` procedure should have its parent as the `Frame` it is called in, not the `Frame` it was defined in.

Use Ok to test your code:

```
python3 ok -q 03
```

Your interpreter should now be able to evaluate the following expressions (and more)!

```
scm> (and)
#t
scm> (and 4 5 (+ 3 3))
6
scm> (and #t #f 42 (/ 1 0))  ; short-circuiting behavior of and
#f
scm> (or)
#f
scm> (or #f (- 1 1) 1)  ; 0 is a true value in Scheme
0
scm> (or 4 #t (/ 1 0))  ; short-circuiting behavior of or
4

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

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

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)

scm> (define f (mu () (* a b)))
f
scm> (define g (lambda () (define a 4) (define b 5) (f)))
g
scm> (g)
20
```

Additional Scheme Tests (1 pt)

Your final task in Part I of this project is to make sure that your scheme interpreter passes the additional suite of tests we have provided.

> To run these tests (worth 1 point), run the command:
>
> ```
> python3 ok -q tests.scm
> ```
>
> If you added any `(exit)` commands outside of the optional section in this file, make sure to remove them so that all the tests are run! You should not have to remove any of the provided `(exit)` commands in the optional section. **The best way to check that you've passed is to use the score command in ok.**

If you have passed all of the required cases,

> you should see 1/1 points received for `tests.scm` when you run `python ok --score`. If you are failing tests due to output from `print` statements you've added in your code for debugging, make sure to remove those as well for the tests to pass.
>
> One you have completed Part I, make sure you submit using OK to receive full credit for the checkpoint.
>
> ```
> python3 ok --submit
> ```
>
> If you'd like to check your score so far, use the following command:
>
> ```
> python3 ok --score
> ```

Congratulations! Your Scheme interpreter implementation is now complete!

Part II: 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 built-in 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 editor (https://code.cs61a.org/scheme) and then try it in your own interpreter once you are confident your Scheme code is working. You can also use the web editor to visualize the scheme code you've written and help you debug.

Scheme Editor

As you're writing your code, you can debug using the Scheme Editor. In your `scheme` folder you will find a new editor. To run this editor, run `python3 editor`. This should pop up a window in your browser; if it does not, please navigate to localhost:31415 (localhost:31415) and you should see it.

Make sure to run `python3 ok` in a separate tab or window so that the editor keeps running.

Problem 4 (2 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 '())
()
```

Use Ok to test your code:

```
python3 ok -q 04
```

Problem 5 (2 pt)

Implement the `merge` procedure, which takes in a comparator and two lists that are sorted, and combines the two lists into a single sorted list. A comparator defines an ordering by comparing two values and returning a true value iff the two values are ordered. Here, sorted means sorted according to the comparator. For example:

```
scm> (merge < '(1 4 6) '(2 5 8))
(1 2 4 5 6 8)
scm> (merge > '(6 4 1) '(8 5 2))
(8 6 5 4 2 1)
```

In case of a tie, you can choose to break the tie arbitrarily.

Use Ok to test your code:

```
python3 ok -q 05
```

Extra Credit

> During regular Office Hours and Project Parties, the staff will prioritize helping students with required questions. We will not be offering help with either extra credit problems unless the queue (https://oh.cs61a.org/) is empty.

Problem EC 1 (2 pt)

Modify your interpreter to allow for evaluation that 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.

One way to implement tail recursive behavior is to delay the evaluation of expressions in tail contexts and then evaluate it at a later time. You can do this by wrapping an expression in an `Unevaluated`. An `Unevaluated` is an object that contains all the information needed to evaluate that expression even outside the frame of `scheme_eval`. We would recommend creating an `Unevaluated` class representing an expression that needs to be evaluated in an environment that can then be instantiated to encapsulate this information.

You will then have to modify your `scheme_eval` function to:

1. Determine whether or not an expression is in a tail context and create `Unevaluated`s as appropriate
2. Handle evaluation of `Unevaluated`s if one is passed in to `scheme_eval`

You should not change the order or types of any of the arguments to `scheme_eval`.

You will likely have to modify other parts of the program besides `scheme_eval` in order to determine which expressions are in tail contexts.

After you have implemented tail recursion, you will need to modify the implementation of `complete_apply`. This function is needed to implement the built-in `apply` procedure, as well as a few other built-in procedures. You may additionally find it useful for your own code.

Currently, `complete_apply` just returns the result of calling `scheme_apply`. However, `complete_apply` differs from `scheme_apply` in that it should never return an `Unevaluated`. Therefore, if `scheme_apply` returns an `Unevaluated`, you should extract and evaluate the expression contained inside the `Unevaluated` instead, ensuring that you do not return an `Unevaluated`.

Use Ok to test your code:

```
python3 ok -q EC
```

Optional Problems

Optional Problem 1 (0 pt)

In Scheme, source code is data. Every non-atomic expression is written as a Scheme list, so 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.

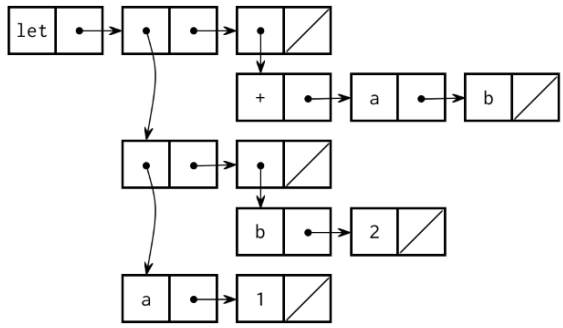
For example, the `let` special form is equivalent to a call expression that begins with a `lambda` expression. Both create a new frame extending the current environment and evaluate a body within that new environment.

```
(let ((a 1) (b 2)) (+ a b))
;; Is equivalent to:
((lambda (a b) (+ a b)) 1 2)
```

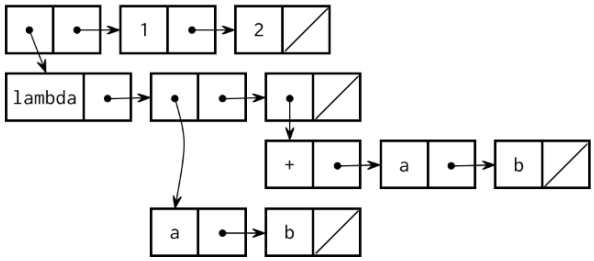
These expressions can be represented by the following diagrams:

Let	**Lambda**

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




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



Use this rule to implement a procedure called `let-to-lambda` that rewrites all `let` special forms into `lambda` expressions. If we quote a `let` expression and pass it into this procedure, an equivalent `lambda` expression should be returned: pass it into this procedure:

```
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! As a reminder, atoms include numbers, booleans, nil, and symbols. You do not need to consider code that contains quasiquotation for this problem.

```
(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>)))
```

Hint: You may want to 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 '())
(() ())
```

Use Ok to test your code:

```
python3 ok -q optional_1
```

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)
     ...))

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

Optional Problem 2 (0 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 procedure 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`, complete the implementation for `do_define_macro`, which should create a `MacroProcedure` and bind it to the given name as in the define form in problem 3. Then, update `scheme_eval` so that calls to macro procedures are evaluated correctly.

Use Ok to test your code:

```
python3 ok -q optional_2
```

Conclusion

Congratulations! You have just implemented an interpreter for an entire language! If you enjoyed this project and want to extend it further, you may be interested in looking at more advanced features, like let* and letrec (http://schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-7.html#%_sec_4.2.2), unquote splicing (http://schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-7.html#%_sec_4.2.6), error tracing (https://en.wikipedia.org/wiki/Stack_trace), and continuations (https://en.wikipedia.org/wiki/Call-with-current-continuation).

Submit to Ok to complete the project.

```
python3 ok --submit
```

If you have a partner, make sure to add them to the submission on okpy.org.