Project 3: Scheme Interpreter

Due Fri Oct 27 at 8pm

Contents

In this project, you will implement an interpreter for a subset of the R5RS Scheme programming language. The main purpose of this exercise is to gain a deeper understanding of the foundational elements of a programming language and how a language operates under the hood. Secondary goals are to write a substantial piece of code in Python and to gain practice with functional language constructs such as recursion and higher-order functions.

The project is divided into multiple suggested phases. We recommend completing the project in the order of the phases below.

Background

An interpreter follows a multistep procedure in processing code. In a program file, code is represented as raw character data, which isn't suitable for interpreting directly. The first step then is to *read* the code and construct a more suitable internal representation that is more amenable to interpretation. This first step can be further subdivided into a *lexing* step, which chops the input data into individual *tokens*, and *parsing*, which generates program fragments from tokens. The end result of this reading process is a structured representation of a code fragment.

Once an input code fragment has been read, the interpreter proceeds to *evaluate* the expression that the fragment represents[1]. This evaluation process happens within the context of an *environment* that maps names to entities. Evaluation is recursive: subexpressions are themselves evaluated by the interpreter in order to produce a value.

Upon evaluating a code fragment, an interactive interpreter will proceed to *print* out a representation of the resulting value. It will then proceed to read the next code fragment from the input, repeating this process.

This iterative combination of steps is often referred to as a *read-eval-print loop*, or *REPL* for short. Interactive interpreters often provide a REPL with a prompt to read in a new expression, evaluating it and printing the result.

In this project, we have provided you with most of the implementation for the read step, though you will fill in a few remaining details in Phase 0. Your primary task, however, will be to implement the functionality needed by the eval step of the interpreter. We have also provided you with an implementation of the print step.

Internal Representations

The parser uses the following representations of Scheme entities:

Scheme Data Type	**Internal Representation**
Numbers	Python's built-in `int` and `float` types.
Symbols	Python's built in `str` type.
Strings	Python's built-in `str` type, where the first and last characters are the double quotes `"`.
Booleans	Python's built in `True` and `False`.
Pairs	The `Pair` class defined in `scheme_core.py`.
Empty List	The `nil` object defined in `scheme_core.py`.

Distribution Code

Start by looking over the distribution code, which consists of the following files:

Lexer and Parser

`buffer.py`	Utility classes for processing input. You should not have to change this file, and you do not need to understand how it works, but you will have to work with `Buffer` objects and thus you should be familiar with the `current()` and `pop()` methods of the `Buffer` class.
`scheme_tokens.py`	Lexer for Scheme. You should not have to change this file, and you do not need to understand how it works.
`scheme_reader.py`	Parser for Scheme. Most of the parser has been implemented for you, but there are a couple of small pieces you must complete, indicated by comments.

[1] An interpreter for an imperative language, such as Python, will *execute* the code fragment if it represents a statement. Scheme, however, only has expressions, so the interpreter only evaluates code.

In completing the parser, you will need to use the `Pair` class that is defined in `scheme_core.py`.

Running `python3 scheme_reader.py` results in an interactive interface that reads in a Scheme expression and prints out a representation of the expression without evaluating it.

Interpreter Core

`scheme_core.py`	The core data structures and logic for the Scheme interpreter. A few basic pieces have been provided for you; examine this code closely and make sure you understand what each function or class does. This file is where you will implement most of the project.
`scheme_primitives.py`	Primitive Scheme procedures that are defined in the global frame in Scheme. Many procedures have been implemented for you. Comments indicate where you will need to add or modify code.

Most of the code you write will be in one of these two files.

Interpreter Driver

`scheme.py`	The top-level driver of the Scheme interpreter, including the read-eval-print loop. You should not have to change anything unless you choose to implement Phase 4.

An input file can be provided at the command line, as in:

```
> python3 scheme.py phase1_tests.scm
```

This reads in each of the expressions in the given file, evaluates them, and prints out the resulting value. Alternatively, if no input file is provided, an interactive REPL is run that reads from standard input.

Test Harnesses

`scheme_test.py`	Testing framework for Scheme programs. You should not have to change this file, and you do not need to understand how it works.
`Makefile`	A Makefile for running test cases.

A testing framework for the interpreter is provided in `scheme_test.py`. The framework takes a Scheme file as a command-line argument, and it uses your interpreter to evaluate each Scheme expression in the file. If the file contains a Scheme comment of the form `; expect value`, the framework compares the result of the expression to the expected value. If the output differs, the framework reports that the test failed. See the provided test files for examples, as well as the documentation at the top of the test harness to see details about its expected input format.

The input file is provided as a command-line argument, as in the following:

```
> python3 scheme_test.py phase1_tests.scm
```

Test Files

`phase1_tests.scm`	Basic tests for Phase 1.
`phase2_all_tests.scm`	Various tests for Phase 2.
`phase2_and_or_tests.scm`	Tests for the `and` and `or` special forms.
`phase2_begin_tests.scm`	Tests for the `begin` special form.
`phase2_define_tests.scm`	Tests for the `define` special form.
`phase2_error_tests.scm`	Error tests for Phase 2.

`phase2_eval_tests.scm`	Tests for the `eval` special form.
`phase2_if_tests.scm`	Tests for the `if` special form.
`phase2_lambda_tests.scm`	Tests for the `lambda` special form.
`phase2_let_tests.scm`	Tests for the `let` special form.
`phase2_letstar_tests.scm`	Tests for the `let*` special form.
`phase2_quote_tests.scm`	Tests for the `quote` special form.
`phase3_tests.scm`	Basic tests for Phase 3.
`phase4_tests.scm`	Basic tests for Phase 4.
`yinyang.scm`	The yin-yang puzzle in Scheme, another test for Phase 4.

The provided tests can be run with the given `Makefile`. It contains the following targets:

- `all`: run all tests for Phases 0-3
- `phase0`, ..., `phase4`: run the tests for an individual phase
- `phase2_all`, `phase2_and_or`, ...: run an individual Phase 2 test, e.g. `phase2_all_tests.scm`, `phase2_and_or_tests.scm`, and so on

Command-Line Interface

Start the Scheme interpreter with the following command:

```
> python3 scheme.py
```

This will initialize the interpreter and place you in interactive mode. You can exit with an EOF (`Ctrl-D` on Unix-based systems, `Ctrl-Z` on some other systems). Later, after you have completed Phase 1, you will be able to exit by invoking the `(exit)` primitive.

If you pass a filename on the command line, the interpreter will take input from the file instead:

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

You can use a keyboard interrupt (`Ctrl-C`) to exit while a file is being interpreted.

Finally, if you use the `-load` command-line argument followed by Scheme filenames, the interpreter will interpret the code in the files and then place you in interactive mode in the resulting environment:

```
> python3 scheme.py -load tests.scm
```

Error Detection

Your interpreter should detect erroneous Scheme code and report an error. The read-eval-print loop we provide you prints an error message when it enounters a Python exception, so it is sufficient to raise a Python exception when you detect an error. It is up to you what information to provide on an error, but we recommend providing a message that is useful for debugging.

Known Departures from R5RS

For simplicity, we depart from the Scheme standard in many places. The following is an incomplete list of discrepancies between our implementation and R5RS:

- We do not support character literals or most of the number formats.
- We are more lenient than the Scheme specification when it comes to identifiers. For instance, tokens such as `+a` or `9c` are treated as valid identifiers.
- The string-literal format implemented by the lexer, as well as the format in which strings are printed, follows the Python rather than the Scheme specification.
- We do not support vectors.
- There is a large set of standard procedures and forms that we do not implement.
- Our implementation of `eval` in Phase 2 differs from the standard.

Though it is not required by the R5RS spec, your implementation must evaluate arguments to a procedure call in left-to-right order.

Phase 0: Parser

Fill in the missing pieces of the Scheme parser in `scheme_reader.py`.

- Modify `scheme_read()` to properly handle quotation markers. In particular, a single quote followed by an expression should result in a new expression that applies the `quote` special form to the following expression:

  ```
  '(1 2)  -->  (quote (1 2))
  ```

  Your parser must also properly handle quasiquotation and both types of unquoting, though your Scheme interpreter is not required to support them. Refer to the Scheme documentation for the syntax of quasiquotation and unquoting. You will also find some tests in the docstring for `scheme_read()` that illustrate the expected behavior of the function.

- Modify `read_tail()` to support dotted pairs. Again, refer to the Scheme documentation for their syntax.

  The argument to `read_tail()` is an instance of the `Buffer` class defined in `buffer.py`. You will need to use the `current()` method, which returns the current input token in the buffer, and `pop()`, which removes the current input token and returns it. (Note that the buffer discards whitespace, since whitespace is not considered an input token.) You should not have to use anything else in `buffer.py`.

  If more than one item appears after the dot, raise an exception as follows:

  ```
  raise SyntaxError("Expected one element after .")
  ```

  You can determine that there is only one item after the dot by reading the next expression and then making sure that the following item in the buffer is the closing parenthesis `")"`.

  There are several tests in the docstring for `read_tail()` that you can look at as examples.

When you are finished, execute the following from the command line to run the integrated doctests:

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

Alternatively, use the Makefile to run the doctests (this leaves out the `-v` flag):

```
> make phase0
```

This will run each of the tests in the docstrings for `scheme_read()` and `read_tail()` and compare the output to the expected output contained in the docstrings.

You will also be able to start an interactive prompt where you can type in Scheme expressions to be parsed:

```
> python3 scheme_reader.py
read> '(hello world)
(quote (hello world))
read> (1 . 2)
(1 . 2)
read> (1 . (2 3))
(1 2 3)
read> (1 . 2 3)
SyntaxError: Expected one element after .
```

You can exit with an EOF (`Ctrl-D` on Unix-based systems, `Ctrl-Z` on some other systems) or with `Ctrl-C`.

Phase 1: Primitives, Environments, and Evaluation

In this phase, you will implement basic features of the Scheme interpreter. Once this phase is complete, your interpreter should be able to evaluate basic Scheme expressions consisting of calls to primitive procedures, as well as compound expressions composed of these basic elements.

Environments

An environment consists of a sequence of frames, each of which binds names to values. Design and implement a representation for frames and environments. Place this code in `scheme_core.py`, and make sure to fill in the `create_environment()` function that creates an an environment with a single empty frame.

Your implementation of frames will need to provide a means of binding a name to a value, both when the name is not in the frame and when it is already bound. In the latter case, the old binding should be replaced by the new one. You may find the Python `dict` type useful.

An environment is comprised of a sequence of frames in some order. You will need mechanisms for binding a name in the first frame and for looking up a name in the sequence of frames. Make sure that when you create a new environment from an existing one, you are not copying frames, since a modification to a frame that is shared by multiple environments is reflected in all of them. You should be able to rely on Python's reference semantics to avoid copying.

Primitive Procedures

Next, modify the `primitive()` and `add_primitives()` functions in `scheme_primitives.py` as needed so that primitive procedures are added to the global frame when the Scheme interpreter is started.

The `primitive()` function is a higher-order function intended to be used as a decorator, as in the following:

```
@primitive('boolean?')
def scheme_booleanp(x):
    return x is True or x is False
```

This is largely equivalent to the following:

```
def scheme_booleanp(x):
    return x is True or x is False
scheme_booleanp = primitive('boolean?')(scheme_booleanp)
```

To make this work, `primitive()` takes in a sequence of names and returns a decorator function. The decorator function takes in a Python function, and for each name that was passed to `primitive()`, it needs to add an object representing a Scheme primitive with the given name and the given Python function as its implementation to the `_PRIMITIVES` list. In the example above, a Scheme primitive with the name `boolean?` and implementation `scheme_booleanp()` should be added to `_PRIMITIVES`.

In order for this to work, you will have to come up with a representation of Scheme primitives that keeps track of the name and implementation of a primitive procedure. We recommend packaging this into an object that is a subtype of the provided `SchemeExpr`, and to place this code in `scheme_core.py`.

Evaluating Symbols

The interpreter code we provide can evaluate primitive values (e.g. numbers and strings), as you can see by examining `scheme_eval()` in `scheme_core.py`. The `scheme_eval()` function is the evaluator of the intepreter. It takes in a Scheme expression, in the form produced by the parser, and an environment, evaluates the expression in the given environment, and returns the result.

You can start the interpreter and type in primitive values, which evaluate to themselves:

```
> python3 scheme.py
scm> 3
3
scm> "hello world"
"hello world"
scm> #t
#t
```

Modify `scheme_eval()` to support evaluating symbols in the current environment. This will allow you to evaluate symbols that are bound to primitive functions:

```
scm> =
[primitive function =]
```

The interpreter printout is dependent on your implementation, and you can implement the special `__str__()` method for your representation of primitives to produce the output you want. You do not have to match the output above.

If a symbol is undefined in the environment when it is evaluated, raise a Python exception, using code such as the following:

```
raise NameError('unknown identifier ' + name)
```

This will be caught by the Scheme read-eval-print loop, and its message will be reported to the user:

```
scm> undefined
Error: unknown identifier undefined
```

Evaluating Call Expressions

Design and implement a process for evaluating Scheme expressions consisting of lists, enabling evaluation of procedure calls. Place this code in `scheme_core.py`. Modify `scheme_eval()` as necessary to run this code when it encounters a list.

A list is evaluated by evaluating the first operand. If the result is a Scheme procedure, then the remaining operands are evaluated in order from left to right, and the procedure is *applied* to the resulting argument values. Special forms have a different evaluation procedure, which we will see in subsequent phases.

If a list is ill-formed (i.e. it does not end with the null list), or if the first operand does not evaluate to a Scheme procedure (or special form in the later phases), your interpreter should raise an exception.

You will need to implement support for applying a primitive procedure to its arguments. This should allow you to evaluate expressions such as:

```
scm> (boolean? #t)
#t
scm> (not #t)
#f
scm> (+ 1 2 3)
6
```

Implementing Primitive Procedures

Implement the remaining primitive procedures in `scheme_primitives.py`. Check the comments for what procedures need to be added, and what their behavior should be. See the implementation of similar procedures for hints on how to write them.

For `apply`, make sure to raise an exception if the first argument does not evaluate to a Scheme procedure.

In order for `procedure?` to work correctly, you will need to implement the `is_scheme_procedure()` function in `scheme_core.py`.

Tests

When this phase is complete, you will be able to run the provided tests for the phase:

```
> make phase1
```

Alternatively:

```
> python3 scheme_test.py phase1_tests.scm
```

Make sure to write your own tests as well, as only a few tests are provided for you.

Phase 2: Special Forms

Extend the evaluation procedure in your interpreter to handle special forms, and implement the special forms below. Except where noted, their behavior should match that in the Scheme specification. Your code to implement this phase should be placed in `scheme_core.py`.

You will need to come up with a representation for special forms that keeps track of the name of the form and its implementation in Python. This is analagous to the representation of primitive procedures in Phase 1. Specifically, we recommend the following:

- Implement a special form as a Python function.
- Define a class for special forms that keeps track of the name and the Python function that implements the form. Modify `is_special_form()` to return true if provided an object of this class.
- Write a decorator for special forms that is similar to the `primitive` decorator in the starter code.
- Modify `scheme_eval()` such that if the first subexpression of a call expression evaluates to a special form, the Python function for handling that special form is called. You will need to pass both the remainder of the call expression and the current environment to this function.

In standard R5RS Scheme, symbols that represent special forms are not reserved. Thus, it is possible to define a variable with the name `if`, `define`, etc. Your interpreter should allow this behavior by defining special forms in the global frame and allowing their names to be redefined in both the global frame and in child frames. You will need to complete the `add_special_forms()` function that will install the special forms in the given environment.

User-Defined Procedures

A user-defined procedure can be introduced with the `lambda` special form. You will need a representation of a user-defined procedure that keeps track of the definition environment (since Scheme procedures are statically scoped), the parameter list, and the body of the procedure. We recommend defining a subclass of `SchemeExpr` that represents user-defined procedures. You will need to modify the `is_scheme_procedure()` function to return true if provided with an object representing a primitive or user-defined procedure.

You only have to implement `lambda`s that take a fixed number of arguments (the first form mentioned in Section 4.1.4 of the Scheme spec).

Evaluating the `lambda` expression itself requires the following:

- Check the format of the expression to make sure it is free of errors. Refer to the R5RS spec for the required format and what constitutes an error.
- Create an object representing a user-defined procedure. Save a reference to the definition environment, the list of parameters, and the body of the `lambda` in this object.
- The resulting value of the `lambda` expression is the newly created procedure object.

You will also need to add support for *applying* a user-defined procedure to an argument list. More specifically, `scheme_eval()` will need to properly handle call expressions where the first subexpression evaluates to a user-defined procedure. The process for applying a user-defined procedure is as follows:

- Evaluate the argument expressions in order from left to right.
- Check that the number of arguments matches the number of parameters required by the procedure.
- Create a new environment that extends the definition environment by a single empty frame.
- Bind the parameter names to the argument values within the context of the newly created frame.
- Evaluate the body in the context of the new environment.

Raise a Python exception if an error is detected in either defining or applying a user-defined procedure.

Derived Forms

The following forms can be implemented by translating them to simpler forms. Do not repeat yourself! If a translation is possible, construct the translated expression and evaluate that rather then repeating code.

Definitions

You are required to implement the first two forms for `define` listed in Section 5.2 of the R5RS spec.

- The first form binds a variable to the value of an expression. You will need to evaluate the expression in the current environment and bind the given name to the resulting value in the current frame. This form cannot be translated into a simpler form.
- The second form defines a function. You only have to handle a fixed number of parameters, so you need not consider the case where the formals contain a period. Make use of the equivalence mentioned in the Scheme spec. Construct the `lambda` expression by appropriately using the `Pair` class, evaluate it, and bind the variable to the result.

You do not have to check that `define` is at the top level or beginning of a body. For this project, the `define` form should evaluate to the name that was just defined:

```
scm> (define x 3)
x
scm> (define (foo x) (+ x 1))
foo
```

Binding Constructs

Implement the `let` form, described in Section 4.2.2 of the Scheme spec. Use the following translation to a `lambda` definition and application:

$$
\begin{array}{l}
(\texttt{let}\ ((name_1\ expr_1) \\
\qquad \ldots \\
\qquad (name_k\ expr_k)) \\
\quad body) \\
\quad \Longrightarrow \\
((\texttt{lambda}\ (name_1\ \ldots\ name_k)\ body) \\
\quad expr_1\ \ldots\ expr_k)
\end{array}
$$

You do not have to implement the "named `let`" form described in Section 4.2.4.

Also implement the `let*` form from Section 4.2.2. This can be translated to `let` using the following recursive rules:

- Base case: if there are no bindings or only one, then `let*` is equivalent to `let`. Thus:

$$
\begin{array}{rcl}
(\texttt{let*}\ ()\ body) & \Longrightarrow & (\texttt{let}\ ()\ body) \\
(\texttt{let*}\ ((name\ expr))\ body) & \Longrightarrow & (\texttt{let}\ ((name\ expr))\ body)
\end{array}
$$

- Recursive case: if there are two or more bindings, then the first is moved to its own `let`, whose body becomes the `let*` minus its first binding:

$$
\begin{array}{l}
(\texttt{let*}\ ((name_1\ expr_1) \\
\qquad\quad (name_2\ expr_2) \\
\qquad\quad \ldots \\
\qquad\quad (name_k\ expr_k)) \\
\qquad body) \\
\qquad \Longrightarrow \\
(\texttt{let}\ ((name_1\ expr_1)) \\
\quad (\texttt{let*}\ ((name_2\ expr_2) \\
\qquad\quad \ldots \\
\qquad\quad (name_k\ expr_k)) \\
\qquad body) \\
)
\end{array}
$$

Other Forms

Implement the following standard forms. Refer to the R5RS spec for their semantics.

- `begin`: This does *not* introduce a new frame, so you cannot translate this to a `lambda`.
- `if`: If the test yields a false value and there is no alternate, then the conditional should evaluate to the predefined `okay` object.
- `and`
- `or`

Quotation

Implement the `quote` form, which merely returns its argument without evaluating it. You do not have to implement `quasiquote`, `unquote`, or `unquote-splicing`.

Non-Standard Forms

Implement the non-standard form

`(eval` *expression*`)`

where *expression* is a valid Scheme expression represented as data. This should evaluate *expression* in the current environment. The following are some examples:

```
scm> (eval '(+ 1 3))
4
scm> (define x 3)
x
scm> (eval 'x)
3
scm> (eval '(+ 1 x))
4
```

Errors

We recommend translating special forms to more fundamental equivalents where possible, to simplify the tasks of error checking and of implementing continuations in the optional Phase 4.

Your implementation of each special form must check for errors where appropriate and raise a Python exception if an error occurs. Examples of errors include a variable definition that is provided more than one expression, a definition with an improper name, a procedure with multiple parameters with the same name, an `if` with less than two arguments, and so on. Specific examples of these:

```
(define x 3 4)
(define 4 5)
(lambda (x y x) 3)
(if #t)
```

Refer to the Scheme documentation for what constitutes erroneous cases for each special form.

Tests

When this phase is complete, you will be able to run the provided tests for the phase:

```
> make phase2
```

You can also run an individual test file for this phase, as in the following:

```
> make phase2_begin
```

Make sure to write your own tests as well.

Phase 3: Tail-Call Optimization

Scheme implementations are required to be *properly tail-recursive*, and they must perform tail-call optimization where possible. We recommend you initially implement your interpreter without tail-call optimization. Once you have the core functionality implemented, you can then restructure your interpreter to support tail-call optimization. You must support it in all contexts required by the Scheme specification.

Proper tail recursion requires that not only does your interpreter use a constant number of Scheme frames for tail-recursive procedures, but that it also use a constant amount of memory in Python. This will require iteratively evaluating tail expressions rather than recursively calling `scheme_eval()`. You will need to do the following to accomplish this:

- Define a class that encapsulates a tail expression with its environment.
- Instead of calling `scheme_eval()` from a tail context of a special form, return an object representing the tail expression and its environment.
- Modify `scheme_eval()` to handle tail expressions. Evaluation will need to be performed in a loop, and encountering a tail expression should repeat the loop with the expression and environment extracted from the tail-expression object. On the other hand, if the result of evaluation is not a tail expression, then `scheme_eval()` should return that result.

You will still need to recursively call `scheme_eval()` in non-tail contexts.

When this phase is complete, you will be able to run the provided tests for the phase:

```
> make phase3
```

Without tail-call optimization, your interpreter will encounter a `RecursionError` on this test due to the recursive calls to `scheme_eval()`. Once you've implemented tail-call optimization, the test should work correctly.

Make sure to write your own tests to ensure that tail-call optimization is applied in all required contexts.

Phase 4 (Optional): Continuations and `call/cc`

This phase will require you to modify `scheme.py`. As such, if you choose to implement it, we recommend making the required modifications in a separate copy of your project, such as a separate branch if you are using git.

A Scheme feature you may implement is continuations, along with the `call-with-current-continuation` special form. In addition, support the `call/cc` shorthand for `call-with-current-continuation`. For this phase, do not implement `values`, `call-with-values`, or `dynamic-wind`.

A continuation represents the entire intermediate state of a computation. When you encounter `call/cc`, your interpreter will need to record the current execution state. This will require backtracking through the execution stack and packaging up the state at each point in a format that will allow you to reconstruct the execution stack whenever the continuation is invoked.

When you build a continuation, the actual call to `call/cc` needs to be replaced by a "hole" that can be filled in when the continuation is invoked. When a continuation is invoked, you should not repeat any computations that have been completed. Thus, the continuation for

```
(begin (display 3) (+ 2 3) (+ 1 (call/cc foo)) (- 3 5))
```

should conceptually represent

```
(begin (+ 1 <hole>) (- 3 5))
```

where the hole is filled in when the continuation is invoked.

After building a continuation, you should immediately resume the newly built continuation, with the hole filled in with a call to the target of the `call/cc` and the continuation object as its argument. In the example above:

```
(begin (+ 1 (foo <continuation>)) (- 3 5))
```

A continuation object can be invoked an arbitrary number of times. It must take a single argument when it is invoked, such as:

```
(<continuation> 2)
```

When a continuation is invoked, the interpreter must abandon the current executation state and resume the invoked continuation instead. The argument of the continuation object fills the hole in the continuation:

```
(begin (+ 1 2) (- 3 5))
```

Abandoning the current execution state requires unwinding the current computation until you reach the read-eval-print loop. (You should support an unbounded number of continuation invocations, so it is not acceptable to recursively call the read-eval-print loop.) Consider using a Python exception to facilitate abandoning the execution state. Once that is done, resume the computation represented by the invoked continuation.

When this phase is complete, you will be able to run the provided tests for the phase:

```
> make phase4
```

You will also be able to run the yin-yang puzzle as follows:

```
> python3 scheme.py yinyang.scm
```

Since this phase is optional, it will not be graded, and it will not be tested on the autograder. Regardless of whether you complete this phase, we recommend turning in a copy of your project that does not contain this phase.

Grading

The autograded portion of this project will constitute 90% of your total score, and the remaining 10% will be from hand grading. The latter will evaluate the comprehensiveness of your test cases as well as your programming practices, such as avoiding unnecessary repetition. In order to be eligible for hand grading, your project must achieve at least half the points on the autograded portion (i.e. 45% of the project total).

Submission

All code that you write for the interpreter must be placed in `scheme_reader.py`, `scheme_primitives.py`, or `scheme_reader.py`. We will test all three files together, so you are free to change interfaces that are internal to these files. You may not change any part of the interface that is used by `scheme.py` or `scheme_test.py`.

Submit `scheme_reader.py`, `scheme_primitives.py`, `scheme_core.py`, and any of your own test files to the autograder before the deadline. **You must submit to the autograder -- pushing code to GitHub does not submit it to the autograder.** We suggest including a `README.txt` describing how to run your test cases.

Acknowledgments

This project is based on the Scheme interpreter project in the *Composing Programs* text by John DeNero.