Project 1: Scheme Interpreter

Due Wed Oct 12 at 8pm

Contents

Warmup 1

Phase 1: Primitives, Environments, and Evaluation 3

Phase 2: Special Forms 4

Phase 3: Tail-Call Optimization 6

Phase 4 (Optional): Continuations and `call/cc` 6

Miscellaneous 7
- Error Detection . . . 7
- Command-Line Usage . . . 7
- Testing Framework . . . 7
- Submission . . . 7

Acknowledgments 7

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.

As part of the code distribution for this project, we have provided you with a lexer for Scheme, most of the parser, and a framework for the interpreter driver. The core structure of the interpreter is entirely up to you, however, and you are free to modify the distribution code in any way you see fit, so long as the interface and behavior of the interpreter remain the same. That said, we expect that you will only have to modify the portions of code noted below.

The project is divided into multiple suggested phases. Though you are free to write it in any order you wish, it will be difficult to test your interpreter incrementally if you do not do so in the order we suggest.

Warmup

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

`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.
`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.
`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.
`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 different methods associated with 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_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.
`phase1_tests.scm`	Basic tests for Phase 1.
`phase2_tests.scm`	Basic tests for Phase 2.
`phase3_tests.scm`	Basic tests for Phase 3.
`phase4_test.scm`	Basic tests for Phase 4.
`yinyang.scm`	The yin-yang puzzle in Scheme, another test for Phase 4.

Then fill in the missing pieces in 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 exceptions 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
```

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.

- 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_frame()` function that creates an 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.

- 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`.

- 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
  True
  ```

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
```

- 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 Phase 2.

  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, 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? true)
  True
  scm> (not true)
  False
  scm> (+ 1 2 3)
  6
  ```

- 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`.

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

```
> 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.

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

  To implement `lambda`, you will need to come up with a representation for user-defined procedures that keeps track of its parameter list and body. You will also need to add support for applying user-defined procedures

to arguments. This should create a new frame, bind the parameters to the argument values in this frame, and evaluate the body in the context of this frame and the definition environment. Scheme procedures are statically scoped, so they need to keep track of the definition environment.

Make sure to raise a Python exception if the number of arguments provided to a Scheme procedure does not match the number of parameters.

Modify the `is_scheme_procedure()` function so that it returns true for both primitive and user-defined procedures.

- `define`: You only have to implement the first two forms in the Scheme specification, and for the second form, you do not have to support a period (i.e. for functions, you only have to support a fixed number of arguments). 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
```

- `begin`
- `if`: If the test yields a false value and there is no alternate, then the conditional should evaluate to the pre-defined `okay` object.
- `and`
- `or`
- `let`: You do not have to support named `let`.
- `let*`
- `quote`
- `eval`: 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.

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. The Python function `add_special_forms()` should accomplish this by adding the special forms to the given frame.

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 true)
```

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

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

```
> python3 scheme_test.py phase2_tests.scm
```

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()`. Instead of calling `scheme_eval()` from the tail contexts of special forms, return a representation of the tail expression and the environment in which it should be evaluated. Then modify `scheme_eval()` to detect when this is the case, replace the current expression and environment with the ones that were returned, and evaluate the new expression in the new environment.

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:

```
> python3 scheme_test.py phase3_tests.scm
```

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`

A Scheme feature you may optionally 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 executation 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.) 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:

```
> python3 scheme_test.py phase4_tests.scm
```

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. However, we will take into account completion of optional phases when assigning final grades to students who fall close to a grade boundary.

Miscellaneous

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.

Command-Line Usage

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
```

Testing Framework

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.

Submission

Place all files for your interpreter directly in the directory `p1` in your EECS 490 GitHub repository. Check in your files and push to GitHub before the deadline.

Acknowledgments

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