Project 3: Scheme Metacircular Evaluator

Contents

https://amirkamil.github.io/project-metacircular-evaluator/

1 Project 3: Scheme Metacircular Evaluator

1.1 *Optional checkpoint due Monday, Mar 8, 2021 at 8pm ET*

1.2 *Final deadline Monday, Mar 15, 2021 at 8pm ET*

In this project, you will implement an interpreter for a subset of the R5RS Scheme programming language, written in Scheme itself. The main purpose of this exercise is to gain further experience with functional programming, including recursion, higher-order functions, functional data abstraction, and message passing. A secondary goal is to achieve a basic understanding of interpretation, as well as the distinction between static and dynamic scope.

The project is divided into multiple suggested phases. The project will need to be completed in the order of the phases below.

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes both code and test cases.

2 Optional Checkpoint

The checkpoint consists of achieving at least 30% of the points on the public and private test cases before the checkpoint deadline. Your grade for the checkpoint will be computed as the better of:

- $min(0.3, score)/0.3$, where $score$ is the fraction of autograded points earned by your best submission before the checkpoint deadline.
- $finalScore$, where $finalScore$ is the fraction of autograded points earned by your best submission before the final deadline.

Thus, completing the checkpoint is **optional**. However, doing it will work to your benefit, since you can guarantee full credit on the 20% of the project points dedicated to the checkpoint.

3 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 are implementing a specific kind of interpreter called a *metacircular evaluator*. This is an interpreter in which the host language (the language in which the interpreter is implemented) is the same as the client language

[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.

(the language that is being interpreted)[2]. The advantage of using the same language for writing the interpreter is that we can rely on the host interpreter to do most of the work. Thus, a metacircular evaluator is often a useful vehicle for exploring language extensions or optimizations, and systems such as PyPy use a similar strategy.

Most of the work in this project entails implementing data abstractions for the internals of an interpreter, as well as functionality involving evaluation, error checking, and special forms. We will rely on the built-in `read` procedure (or optionally, the `read-datum` procedure from the preceding project) for parsing. The result of parsing is Scheme-level representations of the code (in the form of pairs, numbers, symbols, and so on). Our interpreter performs the evaluation step on the resulting structured representations. For simplicity, we only support a subset of R5RS Scheme.

3.1 Distribution Code

Use the following commands to download and unpack the distribution code:

```
$ wget \
  https://amirkamil.github.io/project-scheme-metacircular/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

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

Interpreter

`env.scm`	Implementation of dictionary and environment abstract data types.
`driver.scm`	Top-level driver for the metacircular evaluator. Implements the read-eval-print loop and the `error` procedure for signaling errors.
`primitives.scm`	Implements an abstract data type for primitive procedures, as well as error checking for those procedures.
`core.scm`	Implements the core evaluation procedure, as well as data types for special forms and user-defined procedures.

The code you write will be in `env.scm`, `primitives.scm`, and `core.scm`. You will not modify `driver.scm`, unless you wish to use your Project 2 code for parsing. To do so, place `distribution.scm`, `lexer.scm`, and `parser.scm` in the project directory, and modify the `use-project-parser` global variable in `driver.scm` to be `#t`.

Test Files

`env_tests.scm`	Basic tests for dictionaries and environments (*Phase 1*).
`phase2_tests.scm`	Basic tests for *Phase 2*.
`phase3_all_tests.scm`	Various tests for *Phase 3*.
`phase3_begin_tests.scm`	Tests for the `begin` special form.
`phase3_define_tests.scm`	Tests for the `define` special form.
`phase3_error_tests.scm`	Error tests for Phase 3.
`phase3_if_tests.scm`	Tests for the `if` special form.
`phase3_lambda_tests.scm`	Tests for the `lambda` special form.
`phase3_quote_tests.scm`	Tests for the `quote` special form.
`phase4_tests.scm`	Basic tests for the `mu` form (*Phase 4*).

In addition, the starter files include expected output files (`*.expect`) for the test files for Phases 2-4.

[2] This is related to the concept of a *self-hosting* compiler, which is a compiler that compiles its own source code.

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

- `all`: run all tests for Phases 1-4
- `phase1`, ..., `phase4`: run the tests for an individual phase
- `phase3_all`, `phase3_begin`, ...: run an individual Phase 3 test, e.g. `phase3_all_tests.scm`, `phase3_begin_tests.scm`, and so on

3.2 Command-Line Interface

Start the Scheme interpreter with the following command:

```
$ plt-r5rs driver.scm
```

This will initialize the interpreter and place you in an interactive loop. You can exit with an EOF (`Ctrl-D` on Unix-based systems, `Ctrl-Z` on some other systems).

You can interpret all the code in a file using input redirection:

```
$ plt-r5rs driver.scm < phase2_tests.scm
```

Or:

```
$ cat phase2_tests.scm | plt-r5rs driver.scm
```

3.3 Error Detection

Your interpreter should never crash (except on parse errors if you are using the default `read`). Rather, it should detect erroneous Scheme code and report an error using the provided `error` procedure in `driver.scm`. The procedure takes a string as its first argument, and it optionally takes an arbitrary Scheme datum as a second argument. The arguments are printed, and then the procedure invokes the continuation of the `read-eval-print-loop` to read and evaluate a new expression. The `Makefile` and autograder strip out the actual content of error messages before comparing output. Thus, the contents of an error message are up to you. We recommend providing a message that would be useful for a user.

4 Phase 1: Dictionaries and Environments

We start by building abstract data types for dictionaries and environments. We will build these abstractions using higher-order functions, specifically with message passing and dispatch functions. Messages will be represented as Scheme symbols.

4.1 Dictionaries

R5RS Scheme does not have a built-in dictionary type, so we will construct our own. Implement the `dictionary` higher-order procedure in `env.scm`. The resulting object should support the following messages:

- `length`, with no additional arguments: returns the number of key-value pairs in the dictionary
- `contains`, with a key argument: returns whether the key is contained in the dictionary
- `get`, with a key argument: returns the value associated with the given key

- `insert`, with additional key and value arguments: associates the given key with the given value. If the key is already in the dictionary, this replaces the value associated with the key.

Here are some examples of creating and using a dictionary:

```
> (define d (dictionary))
> (d 'contains 'x)
#f
> (d 'insert 'x 3)
> (not (d 'contains 'x))
#f
> (d 'get 'x)
3
> (d 'insert 'x 4)
> (d 'get 'x)
4
> (d 'length)
1
```

Since different messages require different numbers of arguments, you will need to make use of variadic arguments. Read sections 4.1.4 and 5.2 in the R5RS spec for how to define variadic procedures.

You will need to use mutators to store and modify key-value pairs. You may use the `set!`, `set-car!`, and `set-cdr!` procedures within your dictionary implementation. Refer to the R5RS spec for details on these procedures.

You may implement your dictionary using any underlying representation. You do not have to use a hashtable or tree-based representation. You may find the built-in `assoc` procedure helpful.

Your dictionary abstraction does not need to perform any error checking.

4.2 Environments

An environment consists of a sequence of frames, each of which binds names to values. A natural representation of an environment is a linked list of individual frames, where each frame has a reference to its parent frame, and the top-level (global) frame has a null parent. Fill in the implementation for the higher-order `frame` procedure using this representation strategy. The resulting object should support the following messages:

- `contains`, with a name argument: returns whether the name is bound in the environment, i.e. in the frame or any of its ancestors
- `get`, with a name argument: returns the value bound to the given name in the closest frame that contains a binding for the name
- `insert`, with additional name and value arguments: binds the given name to the given value in the current frame. If a binding already exists in the current frame, this replaces that binding.

Do not repeat code. Use the dictionary abstraction above to maintain the bindings for an individual frame. You can use the built-in `apply` procedure to forward variadic arguments. The following is an example of using `apply`:

```
> (apply + 1 2 '(3 4))
10
```

Refer to the R5RS spec for more details on `apply`.

We use an empty (null) list to represent a null parent. Refer to the documentation of the `frame` procedure for examples.

Your `frame` implementation does not need to perform any error checking.

When you are done with this phase, run the provided test with:

```
$ make phase1
```

Make sure to write additional tests of your own to cover all of the required behavior for dictionaries and environments.

5 Phase 2: Primitives 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.

5.1 Primitive Procedures

Implement `primitive-procedure` in `primitives.scm`. This is a higher-order function that creates a wrapper object around a host-level implementation of a primitive procedure. The wrapper should take a message and any number of arguments. Given the `call` message, an environment, and any number of argument expressions, the wrapper should:

1. Check whether the given number of argument expressions matches the expected number. Use the provided `arity-error` procedure to signal an error. We only support a fixed number of arguments to primitive procedures in this project.
2. Evaluate the argument expressions in some arbitrary order in the given environment, using `scheme-eval` from `core.scm`.
3. Invoke the underlying host implementation on the resulting argument values.

Do not repeat or write unnecessary code. Use the built-in `map` procedure for performing the map pattern rather than reimplementing it.

Refer to `add-primitives` for which primitive procedures we support in our interpreter.

5.2 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 `core.scm`. This procedure 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:

```
$ plt-r5rs driver.scm
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> =
#<procedure:...n/primitives.scm:68:2>
```

The line number in the printout is dependent on your implementation. You do not have to match the output above.

If a symbol is undefined in the environment when it is evaluated, signal an error by invoking the `error` procedure defined in `driver.scm`. This will print the provided message and optional additional argument and return the interpreter back to the REPL, as described in *Error Detection*. The following is an example of the resulting behavior:

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

You do not need to match this error message.

5.3 Evaluating Call Expressions

Modify `scheme-eval` so that it can evaluate a call expression (combination), which is comprised of a list. A list is evaluated by recursively evaluating the first operand. The result must be a callable object. In this project, all callable objects are represented by host procedures, so you should use the built-in `procedure?` predicate to check the result of evaluating the operand. If the result is a callable, then pass it the `call` message, the environment, and the remaining list elements.

If the first operand does not evaluate to a host procedure, your interpreter should signal an error with the `error` procedure.

You should now be able to evaluate calls to primitive procedures, such as:

```
scm> (boolean? #t)
#t
scm> (not #t)
#f
scm> (+ 1 (* 2 3))
7
```

5.4 Division Procedures

As mentioned in *Error Detection*, our interpreter is responsible for all error checking. This includes type checking for procedures that expect specific types. Take a look at the provided `type-checked` procedure, which wraps the implementation of a primitive with an adapter that performs type checking. Refer to `add-primitives` for how `type-checked` is used.

Division procedures not only require their arguments to be numbers, but they also require the second argument to be nonzero. Implement the `divide-checked` procedure, which wraps a division procedure with an adapter that does both type checking and value checking. Make use of the provided `check-arg-types` where appropriate.

5.5 Tests

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

```
$ make phase2
```

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

6 Phase 3: Special Forms

Extend the evaluation procedure in your interpreter to handle special forms. A special form differs from a procedure in that it expects its arguments unevaluated, operating on the arguments as data instead.

First, complete the `special-form` wrapper in `core.scm`. This should be similar to `primitive-procedure`, but significantly simpler. Upon receiving the `call` message, a special-form object should invoke the underlying implementation with the subsequent arguments.

Next, proceed to actually implement the required functionality for the following forms. Fill in the `scheme-*` procedures (e.g. `scheme-begin`) to perform the correct actions for each individual form. Refer to the R5RS spec for full details about each form, including what constitutes an error.

- Implement the `begin` form. This should just evaluate each of its arguments in order, and the result of the `begin` should be the result of the last argument expression.
- Implement the `if` form. If the test yields a false value and there is no alternate, the result is unspecified. You do not need to do anything special for this case. (In other words, rely on the unspecified value produced by the host interpreter's implementation of `if` or other forms.)
- Implement the `quote` form, which simply returns its argument unevaluated.

6.1 User-Defined Procedures

A user-defined procedure can be introduced with the `lambda` special form. Start by implementing `lambda-procedure`, which returns an object representing a user-defined lambda procedure. This object keeps track of the formal parameters, body, and definition environment of a user-defined procedure – lambda procedures are statically scoped, so we need access to the definition environment when the procedure is actually called.

The object returned by `lambda-procedure` should accept the `call` message, accompanied by an environment argument and the arguments to the procedure call. Given the `call` message, it should:

1. Check whether the given number of arguments matches the expected number. Keep in mind that internally, the procedure object also receives the environment as its first argument, in addition to the arguments provided by the user.
2. Evaluate the arguments in some arbitrary order in the given (dynamic) environment.
3. Extend the definition environment with a new frame.
4. Bind the formal parameters to the argument values in the new frame.
5. Evaluate the body in the new frame.

Do not repeat code. There are similarities to how `primitive-procedure` works, and you may need to do some refactoring to avoid code duplication.

We recommend testing your implementation before moving forward. You can do so by starting `plt-r5rs` (or DrRacket), loading `driver.scm` using the `load` procedure, exiting the interactive loop with an EOF, and then invoking procedures in `driver.scm`, `core.scm`, or `primitives.scm` manually. The following is an example:

```
> (load "driver.scm")
Welcome to the metacircular evaluator!
scm> ^D
Bye.
> (define env (scheme-make-environment))
> (define p1 (lambda-procedure 'test '(x y) '((+ x y)) env))
> (p1 'call env 3 4)
7
```

(continues on next page)

(continued from previous page)

```
> (p1 'call env 3)
Error: incorrect number of arguments to test: expected 2, got 1
scm> ^D
Bye.
(#<void>)
```

Notice that an error puts us back into our interactive loop, since it invokes the continuation for that loop. We can exit with another EOF.

Once you are satisfied that `lambda-procedure` works correctly, proceed to actually implement `scheme-lambda`. You only have to support `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.
- Use `lambda-procedure` to create an object representing a user-defined procedure. You can pass it an arbitrary name, such as `'<lambda>`.
- The resulting value of the `lambda` expression is the newly created procedure object.

6.2 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.
- The second form defines a procedure. You only have to handle a fixed number of parameters, so you need not consider the case where the formals contain a period. Do not repeat code; make use of what you have already written for `lambda`, and refactor if necessary to avoid duplication.

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
```

6.3 Errors

Your implementation of each special form must check for errors where appropriate and invoke the `error` procedure in `driver.scm` 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.

6.4 Tests

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

```
$ make phase3
```

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

```
$ make phase3_begin
```

Make sure to write your own tests as well.

7 Phase 4: Dynamic Scope

To illustrate the value of a metacircular evaluator, we will extend the Scheme language with a `mu` form that creates procedures that use dynamic rather than static scope. Recall that dynamic scope with shallow binding means that the parent environment of the newly created frame for a procedure call is the calling environment, rather than the definition environment. The following is an example of a `mu` procedure:

```
scm> (define y 5)
y
scm> (define foo (mu (x)
                     (+ x y)
                   ))
foo
scm> (foo 3)
8
scm> ((mu (y)
          (foo -3)
       )
       2
      )
-1
```

Here, we have defined `foo` to be a `mu` procedure that refers to a non-local variable `y`. When invoked from global scope, the dynamic environment is the global environment, so `y` evaluates to 5, reflecting its binding in the global frame. However, when `foo` is invoked from within another procedure with its own binding for `y`, then `y` evaluates to the value corresponding to that binding.

Start by implementing an analogue of `lambda-procedure` for procedures with dynamic scope. As always, avoid repeating code, refactoring if necessary.

Second, implement a `scheme-mu` procedure that handles evaluation of a `mu` expression, resulting in a newly created procedure object.

Finally, modify `add-special-forms` so that it adds a binding for the `mu` special form to the given environment.

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

```
$ make phase4
```

Make sure to write your own tests as well.

8 Rules and Regulations

The goals of this project include obtaining experience in functional programming. As such, your code is subject to the following constraints:

- You may not use any non-R5RS-standard code.
- You may not use the built-in `eval` procedure, or similar procedures, except in testing.
- You may not use any procedures or constructs with side effects. Included in the prohibited set are any mutators (procedures or constructs that end with a bang, e.g. `set!`), and you may only use `define` at the top level (i.e. at global scope). The only exception to this rule is that you may use `set!`, `set-car!`, and `set-cdr!` in `env.scm`.
- You may not use any iteration constructs. Use recursion instead.

To facilitate checking the rules above, the following symbols may not appear in `env.scm`, `primitives.scm`, or `core.scm`:

- `read`
- `eval`
- `current-input-port`
- `current-output-port`
- `open-input-file`
- `open-output-file`
- `with-input-from-file`
- `with-output-to-file`
- any symbol ending with `!`, except `set!`, `set-car!`, and `set-cdr!` in `env.scm`
- `define` as the first item in a list, except at global scope
- `peek-char`
- `read-char`
- `do`
- `for-each`
- `syntax-rules`

Any violations of the two sets of rules above will result in a score of 0.

In addition, the standard Engineering Honor Code rules apply. Thus, you may not share any artifact you produce for this project outside of your partnership, including code, test cases, and diagrams. This restriction continues to apply even after you leave the course. Violations will be reported to the Honor Council.

9 Grading

The approximate grade breakdown for this project is as follows:

- 20% checkpoint
- 70% final deadline autograded
- 10% final deadline hand graded

Hand grading will evaluate the comprehensiveness of your test cases as well as your programming practices, such as documentation, avoiding unnecessary repetition, defining appropriate ADTs, and making proper use of built-in Scheme procedures. In order to be eligible for hand grading, your solution must achieve at least half the points on the autograded, final-deadline portion of the project.

10 Submission

All code that you write for dictionaries and environments must be placed in `env.scm`. The remaining code you write for the interpreter must be placed in `primitives.scm` or `core.scm`. You may not change any part of the interface that is used by `driver.scm`.

Submit `core.scm`, `env.scm`, `primitives.scm`, and any of your own test files to the autograder before the deadline. We suggest including a `README.txt` describing how to run your test cases.

11 Acknowledgments

This project uses elements from the *Structure and Interpretation of Computer Programs* text by Abelson and Sussman, as well as the Scheme interpreter project in the *Composing Programs* text by John DeNero.