Project #3 – Writing a Scheme Interpreter in Scheme
CS 152 Section 6 – Fall 2021
Michael McThrow
San José State University

PAIRS ALLOWED: You may choose to work in pairs on this project. Keep in mind that once you begin working as partners, you may not change partners during the duration of the project, though you may either choose a different partner or work alone in the next project that allows pairs.

Introduction

Your task for Project #3 is to write an interpreter for Scheme using Scheme in DrRacket. Your interpreter will be called `my-scheme.scm` and it will be written in R5RS Scheme just as Project #2. (Make sure your program begins with the line `#lang r5rs` to ensure it is a R5RS Scheme program.) An interpreter written in the same language that is to be interpreted is known as a *metacircular interpreter*. For example, if we wrote a Python interpreter in Python, that interpreter would be metacircular.

Your task is to define the function `eval-prog`, which evaluates a list of S-expressions while maintaining the environment(s) that each S-expression needs. The function `eval-prog` returns the evaluated result of the final S-expression in the input list.

For example, if we call `(eval-prog '((+ 1 2 3 4 5)))`, it should return `15`. Calling `(eval-prog '((define x 10) x))` should return `10`.

The beauty about writing a Scheme interpreter in Scheme is that we don't have to worry about parsing S-expressions. In fact, we don't have to write any string-handling code at all in our evaluator. Instead, we can simply leverage Scheme's built-in support for handling S-expressions.

It is your choice regarding how to implement the environment (e.g., cons pairs, lists, binary search trees, etc.) as long as they adhere to functional programming style (i.e., no mutation and no side effects). You may want to look into R5RS Scheme's vector data type, which may be more efficient than list-based data structures for environments.

Remember, <u>we will not be using mutation in this program</u>. The challenge of implementing an interpreter in functional programming style is not using mutation for updating our environments.

Part 1: Evaluating Primitive Types (20% of maximum points)

Your first milestone is to make sure that you can evaluate very simple expressions that are not function calls or are not syntactic sugar of function calls (e.g., `'(1 2 3)` is syntactic sugar for `(cons 1 (cons 2 (cons 3 ())))`, which is a function call). For example, `(eval-prog '(5))` should return `5`. The same applies to boolean values `#t` and `#f`. If the expression that you are evaluating is a symbol, perform an environmental lookup. (For testing purposes, you may want to build test environments that can be used while you don't have functions like `define` and `set!` working.) If the symbol resolves to an expression, return the value of that expression. If the symbol

resolves to a built-in function, then return `'BUILT-IN-FUNCTION`. If the symbol resolves to a non-built-in procedure, then return `'PROCEDURE`.

Part 2: Primitive Functions (80% of maximum points)

Your task is to implement all of the primitive functions listed in Appendix A. Here are the weights of the following implementations:

Function/Function Class	Percentage of Maximum Points
define	10%
lambda	10%
set!	10%
cons, car, cdr	10%
if	5%
quote	5%
+, -, *, /, remainder, modulo, and, or	7.5%
eq?, equal?, =, >, <, >=, <=, =, empty?, number?, pair?, symbol?	7.5%
The ability to call functions, whether built-in or program-defined.	15%

Once the above has been implemented, we can run Scheme programs such as the following:

```
(define (even? n)
  (= (remainder n 2) 0))
(even 10)

(define (length elems)
  (if (empty? elems) 0 (+ 1 (length (cdr elems)))))
(length (cons 1 (cons 3 (cons 5 (quote ())))))
```

Return Values

- `eval-prog` returns the evaluation of the final S-expression in the S-expression list. For example, if I run

  ```
  (eval-prog '((define x 10) (set! x (+ 2 3)) (* x x)))
  ```

  it should return `25` as its answer.

Grading Nodes

- If your code does not run due to syntactic errors, then that relevant portion of the code gets a zero grade.
- Your program must begin with the line `#lang r5rs`.
- Your function `eval-prog` must have the parameters in the exact order as specified; do not change the number of parameters, their order, or their expected types, or they won't get graded.
- Your submission must be a single file called `my-scheme.scm`. Do not split your work into separate files.
- Mutation is banned, even when implementing set! in your interpreter. If you use `set!`, `set-car!`, `set-cdr!`, or any other function that uses mutation, then that code gets a zero grade.
- Do not use Scheme's `eval` function to implement any portion of this assignment, or you won't get credit for those portions.
- Please refrain from defining custom macros in your source code.

Tips

- **START EARLY!** This may turn out to be the toughest assignment in this course, so allow yourselves plenty of time to complete this interpreter. I highly recommend working with a partner on this assignment.
- I highly recommend building a collection of test cases that you can use to test your Scheme interpreter.
- Remember that any expression in the dialect of Scheme defined in this project is a valid R5RS Scheme expression.
- **Approved References:**
  - Feel free to use Sections 3.2 and 4.1 of *Structure and Interpretation of Computer Programs*, which discusses how environments work and how a basic Scheme interpreter works. It is okay to use this reference. Please keep in mind that your environment will not use mutation, whereas in the textbook the environment it uses is modified using mutation.
  - I am also okay with you using Peter Norvig's Java (https://norvig.com/jscheme.html) and Python (https://norvig.com/lispy.html) implementations of Scheme interpreters as a reference. Once again, these implementations have the same caveat as the one in SICP.
  - For the curious who is really interested in learning more about the history of Lisp, I highly recommend taking a read of the first 13 pages of *The LISP 1.5 Programmer's Manual* (http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf). Page 13 has a description of a basic LISP 1.5 interpreter, which Alan Kay calls "the Maxwell equations of software." Please keep two things in mind:
    - LISP 1.5 is a very early version of Lisp from 1962 that has different semantics from Scheme (particularly when it comes to variable scoping) or from other modern Lisps such as Common Lisp and Clojure.
    - The syntax of page 13 is not our familiar S-expressions, but an earlier syntax known as M-expressions. It turns out that when John McCarthy created Lisp in the late 1950s, he originally meant to have Lisp implemented using M-expressions; S-expressions were originally meant as primitive representations of Lisp expressions. However, when one of his students read his paper and implemented Lisp, he used S-expressions. The other researchers in McCarthy's lab also wrote their Lisp programs in S-expressions, and S-expressions ultimately became the canonical way of writing Lisp programs. However, there have been attempts at making Lisp-style languages that don't use S-expressions; in

the 1990s Apple worked on an object-oriented Lisp called Dylan which syntax resembles contemporary procedural programming languages. Julia, which is growing in popularity among data scientists, is heavily inspired by Lisp yet also eschews S-expressions, and some have liked JavaScript to Lisp.

- If you do study any approved references, cite them in your code comments.
- Please don't refer to any other Scheme or Lisp implementations, however.

Appendix A: Scheme Primitive Functions

Name	Usage	Definition
`define`	`(define name expr)` `(define (function-name x1 … xN) body)`	If the second element is a symbol, then evaluates `expr` and assigns it to name in the current environment. If the second element is a list, then this is syntactic sugar for `(define function-name (lambda (x1 … xN) body))`
`lambda`	`(lambda (x1 … xN) body)`	Creates an anonymous function that evaluates `body` with parameters `x1` to `xN`, or with no parameters. Note that body can consist of multiple expressions `expr1` to `exprM`. The result of `body` is the value of the final expression `exprM`.
`set!`	`(set! name expr)`	Evaluates `expr` and modifies the value stored at `name` to be the evaluated result of `expr`.
`cons`	`(cons expr1 expr2)`	Creates a cons cell where the first element contains the evaluated result of expr1 and the second element contains the evaluated result of expr2.
`car`	`(car expr)`	Returns the first element of the cons cell expr1. If expr does not evaluate to a cons cell, then return an error.
`cdr`	`(cdr expr)`	Returns the second element of the cons cell expr1. If expr does not evaluate to a cons cell, then return an error.
`quote`	`(quote expr)`	Returns expr without evaluating it.
`+`	`(+ x1 … xN)`	Performs the sum x1 + … + xN. If there is only one argument x1, it returns x1.
`-`	`(- x1 … xN)`	Performs the repeated difference x1 - … - xN. If there is only one argument x1, then it negates it.
`*`	`(* x1 … xN)`	Performs the product x1 * … * xN. If there is only one argument x1, it returns x1.
`/`	`(/ x1 … xN)`	Performs the repeated division x1 / … / xN. If there is only one argument x1, it returns 1 divided by x1. Returns an error if any of the parameters evaluate to 0.
`remainder`	`(remainder x y)`	Returns the remainder of the division x divided by y.
`modulo`	`(modulo x y)`	Returns the modulo of the division x divided by y. There is a nuance between remainder and modulo that is important when dealing with negative numbers.

`and`	`(and x1 … xN)`	Performs x1 ∧ … ∧ xN. Returns #f at the first occurrence of a parameter that evaluates to #f.
`or`	`(or x1 … xN)`	Performs x1 ∨ … ∨ xN. Returns #t at the first occurrence of a parameter that evaluates to #t.
`if`	`(if expr1 expr2 expr3)`	If expr1 evaluates to #t, then evaluate expr2 and return it. Otherwise evaluate expr3 and return it.
`=`	`(= expr1 expr2)`	If expr1 and expr2 both evaluate to numbers, then this function checks to see if they are equal. Returns #t if equal, #f otherwise. Returns an error if expr1 or expr2 don't evaluate to numbers.
`>`	`(> expr1 expr2)`	If expr1 and expr2 both evaluate to numbers, then this function checks to see if expr1 > expr2. Returns #t if equal, #f otherwise. Returns an error if expr1 or expr2 don't evaluate to numbers.
`<`	`(< expr1 expr2)`	If expr1 and expr2 both evaluate to numbers, then this function checks to see if expr1 < expr2. Returns #t if equal, #f otherwise. Returns an error if expr1 or expr2 don't evaluate to numbers.
`>=`	`(>= expr1 expr2)`	If expr1 and expr2 both evaluate to numbers, then this function checks to see if expr1 >= expr2. Returns #t if equal, #f otherwise. Returns an error if expr1 or expr2 don't evaluate to numbers.
`<=`	`(<= expr1 expr2)`	If expr1 and expr2 both evaluate to numbers, then this function checks to see if expr1 <= expr2. Returns #t if equal, #f otherwise. Returns an error if expr1 or expr2 don't evaluate to numbers.
`eq?`	`(eq? expr1 expr2)`	Evaluates the expr1 and expr2 and checks if they both evaluate to symbols of the same name. Note that symbols are case-sensitive. Returns #t if they do, #f otherwise.
`equal?`	`(equal? expr1 expr2)`	Encompasses = and eq? for numbers and symbols, respectively, and also checks cons cells for equality.
`empty?`	`(empty? expr)`	Returns #t if expr evaluates to the empty list, #f otherwise.
`number?`	`(number? expr)`	Returns #t if expr evaluates to a number, #f otherwise.
`pair?`	`(pair? expr)`	Returns #t if expr evaluates to a cons cell, #f otherwise.
`symbol?`	`(symbol? expr)`	Returns #t if expr evaluates to a symbol, #f otherwise. Note that the Boolean constants #t and #f are not considered symbols, but rather constants of the Boolean type.