Technical Report TR-20230505-1

Scheme Interpreter for R
Project Report

Jesse Mitchell
Advisor: Dr. Dana Vrajitoru
Indiana University South Bend
2023-05-05

Abstract

For this project, I wrote a Scheme interpreter in R, which allows programs written in it to call R functions directly and to be exported as R functions themselves. In addition to commands to evaluate strings as Scheme expressions directly or run entire files of Scheme code, it includes a fully functional read-eval-print loop (REPL). The interpreter includes full support for tail call optimization, which was accomplished through a sort of just-in-time compilation that converts Scheme expressions into single-argument R functions and the creation of a simple virtual machine that implements a modified version of the trampoline pattern using those functions, with Scheme environments as virtual stack frames. This paper also includes a partial list of the syntactic forms implemented in the interpreter's version of Scheme, including support for the creation of macros through the `define-macro` command.

1. Introduction

For my project, I created a Scheme interpreter that runs within R and allows programs written in it to directly call R functions.

Scheme is a dialect of Lisp created in 1975 by Gerald Sussman and Guy Steele, drawing influences from ALGOL 60 and lambda calculus (Sussman and Steele 1975). Like most Lisp dialects, it's a declarative functional language. Unlike most Lisp dialects at the time, it features lexical scoping, meaning that functions look for variable bindings in the environments in which they were defined rather than in the environments from which they were called.

R is a language primarily used for statistics and data analysis, created by Ross Ihaka and Robert Gentleman in 1993 as an open-source version of the language S, with similar lexical scoping rules to Scheme (Peng 2022). It also has an extremely large library of third-party packages that can be downloaded and installed with the `install.packages` function, implementing features such as advanced 2d and 3d data plotting, creating different types of machine learning models, or plotting the ROC curves of binary classifiers. This library is, in fact, the main reason that I chose R as the underlying language for my interpreter.

This project was motivated by two main reasons. First, one of the defining features of Scheme is tail call optimization, which allows certain types of recursive functions to be executed with constant space complexity and thus, without having to worry about stack overflows. This makes it easier to represent some mathematical concepts as programs. This is particularly useful in a math-heavy language such as R. Second, writing an interpreter gave me a deeper understanding of how computer programs work at a basic level. In particular, figuring out my implementation of tail call optimization involved creating what was essentially a simplistic virtual machine and manually handling its call stack. My interpreter takes the form of an R package including, among other things, a function that launches a basic read-eval-print loop (REPL) and other functions that evaluate strings of text and text files containing Scheme code.

2. The Lexer and Parser

The first two components of any compiler or interpreter are the lexer and the parser; the former converts raw text into a series of tokens, and the latter uses those tokens to build data structures representing the program's code. Both are fairly straightforward in this case, as every expression in Scheme is essentially a direct representation of its own abstract syntax tree. Anything surrounded by double quotes is a string, and anything else follows a few simple rules:

- Comments start with a semicolon and end with a line break
- Lists are surrounded by parentheses, with elements separated by whitespace
- Pairs and improper lists (see the section "Pairs and Lists", below) have a period before the final element
- Vectors are the same as lists, but preceded by a `#` sign. For example, `#(1 2 3)`. Unlike lists, there is no such thing as an improper vector; for instance, `#(1 2 . 3)` is not a valid Scheme expression
- There are a handful of reader macros, special characters that indicate that whatever follows them should be treated as the second entry in a two-element list whose first entry is determined by the reader macro itself. For example, `,x` is short for `(unquote x)` and `'(1 2 3)` is short for `(quote (1 2 3))`. My implementation of Scheme also allows users to define their own reader macros, as will be explained in more detail later.

My particular implementation of Scheme has some additional rules that evolved as I modified it to better work with R, but I'll address those as they come up in the report. In any case, the above were the rules I had in mind when I initially designed the lexer and parser.

2.1 The First Lexer

My first attempt at making a lexer used the same basic approach as Peter Norvig's simple Lisp interpreter (Norvig 2010a), modified to support strings. The idea was fairly simple: for a first pass, the lexer would take a string of input and use R's `strsplit` function to split it up at every double quote. This would result in a vector of strings, alternating between those that needed to be further split into tokens and those representing literal strings. The latter didn't need much further processing, just having to convert escaped characters into their literal forms (`\n` into a line break, for example), while the former would require a few passes of string substitution:

- convert parentheses and reader macros into versions surrounded by whitespace
- remove any substring beginning with a semicolon and ending with a line break
- convert any series of multiple whitespace characters into a single space

As an example, doing this would have converted the following:

```
(define (f x)
  (display 'hello)(newline)
  (+ x 3))
```

into `( define ( f x ) ( display ' hello ) ( newline ) ( + x 3 ) )`. From there, another use of `strsplit` would divide the new string along any whitespace, then I could use the

`Map` function to create a list of the results of passing each term to a function that attempts to turn its argument into a number and, if that fails, turns it into a symbol instead. Splicing the resulting lists and the literal strings all into a single big list would give me the string of tokens I needed, which could then be passed to the parser.

2.2 The Second Lexer

The problem with my first lexer became apparent when I realized that there was no clear precedence between the use of double quotes to denote literal strings and the use of semicolons and line breaks to denote comments. Both had clear, unambiguous precedence over parentheses: a double quote between parentheses still denotes the start of a string, and a semicolon between parentheses still denotes the start of a comment, but parentheses between double quotes are just part of the string, and parentheses between a semicolon and a line break are just part of the comment. However, things are less cut when you compare comments and strings: a double quote in the middle of a comment means nothing, and a semicolon between double quotes is just another character in a string. Which to use depends purely on which character is encountered first, rather than there being clear-cut rules that can be applied to the entire expression in order. Consider the following:

```
(this is ;unfortunately "not" quite
 correct)
```

Splitting it along the double quotes first and then applying the other rules will, once the lexer and parser have both finished, result in the list `(this is "not" quite correct)` rather than the desired list `(this is correct)`, as the substring containing the semicolon ends before the word `"not"`. Conversely, attempting to parse for comments before strings means that the following:

```
(a b "this is a sentence; it contains a semicolon"
 c d e)
```

will start by removing everything from the semicolon on in the first line, including the double quote. Parsing the result will yield the list `(a b "this is a sentence c d e)")`, assuming the parser returns upon running out of tokens rather than throwing an error because its closing right parenthesis is missing.

To address this problem, I turned to the package `rly`, which contains R implementations of the tools `lex` and `yacc`, used to make lexers and parsers respectively. While the parser I had written was both simple enough and functional enough that modifying it to work with a new lexer was more worthwhile than trying to start from scratch with `yacc`, redoing the lexer with `lex` was easier than trying to reinvent the wheel and come up with my own way to process strings into tokens that would work with rules that had equal precedence.

The second lexer is actually much more straightforward than the first: each token type consists of a regular expression and a function. Any text that matches the regular expression for a given type is converted into a token of that type, which is then passed to the function for processing. For example, processing literal strings involves running them through the same transformations as in the first lexer, and processing complex numbers involves converting them into `a+bi` format (my interpreter also accepts values like `2i` or `13+i`, which would be converted into `0+2i` and `13+1i` respectively) and then passing them to `as.complex` for conversion into actual numbers.

2.3 The Parser

Due to Scheme's extremely uniform syntax, the parser isn't particularly complicated. It consists of two functions, `lex.next` and `lex.reader`. As the name might imply, `lex.next` queries the lexer and generates the next term. This usually just means returning the next token, but there are a few exceptions. If the next token is a reader macro, the next term will be a list consisting of the symbol corresponding to that macro and the term after that (retrieved with a recursive call to `lex.next`). If the next token is a left parenthesis, the next term will be a list, generated by a call to `lex.reader`. The opening tokens for vectors and mutable lists work the same way, only the result of the call to `lex.reader` is passed to either `as.svector` or `as.mlist` before being returned.

The other half of the parser is `lex.reader`, which assembles the terms from `lex.next` into lists. It works by repeatedly calling `lex.next` for more terms and adding them to a list until it either runs out of terms or hits a right parenthesis, right curly bracket, or right square bracket. When that happens, it either returns or, if the symbol it encountered isn't the one it was expecting (for example, a right square bracket to end a list that was started with a left parenthesis), throws an error.

2.4 Scheme Data Types

Most of the atomic data types (that is, those that can't be further subdivided in a meaningful way) in Scheme are the same as in any other programming language. Strings are sequences of characters, numbers are values that can be used to perform arithmetic, procedures are values that do things when called with other variables as arguments, and so on. One difference between Scheme and most other languages, including other versions of Lisp, is that while any data type can be passed to logical operators like `if` or `not`, only the false boolean value (which, in my implementation, can be written as `#f` or `false`, both of which are case-insensitive) counts as false. Any other values, including the empty list and numerical values equal to zero, count as true.

Another Scheme data type not found in many other languages, though it is found in other versions of Lisp and in R, is the symbol. Symbols are similar to strings, but generally represented without quotation marks; `"cat"` is a string, for example, and `cat` is a symbol. There are functions for converting symbols into strings and vice versa, but symbols are generally just unique identifiers with descriptive names. R and Scheme both use them internally as variable names, and they can also be used in the same way as enumerations in other languages.

2.5 Lists and Pairs

By far the most important data structure is the list. In the usual representation, a list is surrounded by parentheses and the items within it, which can be of any type, are separated by whitespace. For example, `(1 "cat" (2 3) apple)` is a list consisting of the number `1`, the string `"cat"`, the list `(2 3)`, and the symbol `apple`. The empty list, represented by just `()`, is the only list that also counts as an atom. Because code can often include deeply nested lists, I've also made it possible for lists to be surrounded by square brackets instead of parentheses, to make it easier to keep track of which ones match up. For instance, it's easier to tell where each nested expression in `(let ([a (f (g [h x]))]) ...)` begins and ends than it would be if it was written as `(let ((a (f (g (h x))))) ...)`.

Even simpler than lists are pairs, also known as `cons` (short for "construct") cells after the command used to create them. A pair consists of two items, which can be any type of object including other pairs. For historical reasons dating back to the first implementation of LISP on the IBM 7090 (McCarthy et al. 1962), the first item in each pair is known as the `car` (short for "Contents of the Address Register") and the second is known as the `cdr` (short for "Contents of the Decrement Register"). These names are also shared by the functions that access each half of a pair, and because the functions are so commonly composed with one another, Scheme includes a sort of shorthand for the combinations: any function whose name is a `c` and `r` with some combination of `a` and `d` in between is a composition of `car` and `cdr` in the same order as the `a` and `d` terms. For example, `caddr(x)` is equivalent to `car(cdr(cdr(x)))`. My version of Scheme, like most, includes functions of this sort going from `caar` to `cddddr`.

Pairs are represented almost the same way as lists, except that there's a period between the two terms. For instance, a pair whose `car` is 1 and whose `cdr` is 2 would be represented as `(1 . 2)`. If the `cdr` of a pair is itself a pair, its parentheses and the period preceding it can be omitted. For example, `(1 . (2 . 3))` could also be written as `(1 2 . 3)`. The same applies to the empty list, so `(a . ())` could just be written as `(a)`. This is because in most versions of Scheme and Lisp, lists are implemented as linked lists with pairs as the nodes, with the `car` of each pair being the value of an item in the list and the `cdr` being a pointer to the next pair, except for the final element in the list, whose `cdr` is just the empty list. Thus, the list `(1 2 3 4)` is in fact the series of pairs `(1 . (2 . (3 . (4 . ()))))`. It's also perfectly legal, though less common, to omit the dots and parentheses from only some of the pairs in the list, meaning the above could also be written as `(1 . (2 3 4))` or `(1 2 . (3 4))`.

A list whose final `cdr` is something other than the empty list, such as `(1 2 3 4 . 5)`, is known as an improper list. While lists in R are implemented differently, and my interpreter stores its lists in the R format for ease of interoperability with native R functions, improper lists are useful enough in some parts of Scheme that I actually found a way to emulate them: when an improper list is created, either through direct user input or as a result of calls to `cons`, the final entry is stored inside an object of the R class `dotted.tail`.

Unlike Scheme, R allows items in lists to have names. As these are used by the `do.call` function to pass named arguments to R functions, this was a feature I felt it necessary to include, which I did by borrowing the syntax that the language Racket uses for keyword arguments (Schwarzer 2023). For example, the list created by the command `list(a=1,b=2)` would be represented in my version of Scheme as `(#:a 1 #:b 2)`, and the function call `seq(1,10,length.out=4)` would be written as `(seq 1 10 #:length.out 4)`.

2.6 Mutable Data Structures

Pairs (and, by extension, lists) in most versions of Scheme are also mutable, meaning that it's possible to change the `car` or `cdr` of an existing pair using the commands `set-car!` and `set-cdr!`, without changing the address of the pair itself in memory. This can lead to convoluted self-referential structures, as there's nothing stopping users from setting the `car` or `cdr` or a pair to itself, and Scheme in fact has a specific notation for when that happens: a pair or list which contains itself is written in the form `#n=...`, where n is an integer indicating the number of self-referential structures that have already appeared in the current expression and `...` is the way the pair or list would normally be represented. In this representation, any further instances of the pair or list are replaced with `#n#`. For example, a pair whose `car` is 23 and whose `cdr` is itself would be represented as `#0=(23 . #0#)`, and a pair whose `car` is itself and whose `cdr` is the aforementioned pair would be represented as `#0=(#0# . #1=(23 . #1#))`.

Lists in R are, of course, not mutable. R does include a data structure called a pairlist that works the same way internally as traditional Scheme or Lisp lists, but there's no user-accessible way to directly change the `car` or `cdr` of a pairlist. When I did so manually (by using the Rcpp package to directly access the necessary C functions in the R API), I discovered that both improper lists and self-referential lists frequently cause R to freeze and/or crash. However, because mutable pairs and lists are essential to implementing many useful Scheme features, I took another cue from Racket (Flatt and PLT 2023b) and created a separate mutable pair data type called `mpair`, which can be used to construct mutable lists. As in Racket, these are printed the same as regular lists except that they use curly braces instead of parentheses. Unlike in Racket, they can also be input directly; typing `{1 2 3}` into the REPL will evaluate to a mutable list of the numbers 1 through 3.

Because environments are the only mutable types in R, each mutable pair is simply an environment containing the variables `car` and `cdr`, with its class set to `mpair`. Mutable pairs and lists are created with the `mcons` and `mlist` functions rather than `cons` and `list`, but the rest of the functions for dealing with lists (`car`, `cdr`, `cadr`, and so forth) are generic functions that use R's S3 class system to call more specific counterparts; `car(x)`, for example, will call either `car.default(x)` or `car.mpair(x)` depending on whether or not `x` is a mutable pair.

The other mutable data structure in Scheme are vectors, which are similar to arrays in C or Java: they support random access, and each member can be referenced by its index, with the first being at position `0`. Being mutable, they are of course also implemented as environments with a custom class identifier. Values are read from vectors with the command `(vector-ref vector index)` and assigned with the command `(vector-set! vector index value)`.

2.7 Scheme Output

For the most part, values in Scheme are output in the same format in which they would have been entered by a user. The main exceptions are that lists and symbols entered into the REPL would need to be preceded by an apostrophe to stop them from being evaluated (see the section on Eval and Apply below) and self-referential structures like `#0={1 . #0#}` can't be entered directly as input. There are, however, some data types that can't be directly printed in a human-readable format, such as I/O ports or functions. When printing such cases, Scheme will typically default to a placeholder in the format `#<class>`, where `class` is the type of the object, such as `#<input-port>` or `#<function>`. This is generally all the information provided, though I have been able to add more information in a few specific cases. Scheme procedures that have been defined with names (as opposed to those created directly with `lambda` expressions) will include those names when printed, such as `#<procedure:square>`. Similarly, promises will display either the expression they're meant to evaluate or the result of the evaluation, depending on whether they've been forced yet. For example, the promise to evaluate the expression `(+ 2 2)` will display as `#<promise:(+ 2 2)>` before being evaluated and `#<promise!4>` afterward.

To display its output, my interpreter uses the function `scmprint`, which also uses R's class system to handle different types of output. When dealing with lists, mutable pairs, vectors, and promises, it starts by calling the function `mapcycle`, which searches through all four types for self-referential structures using the following algorithm:

1. It creates an empty list to serve as a repository of all the self-referential terms it finds;
2. It then calls the internal function `explore`, with its first argument being the term it wants to examine and the second, which it will use as a list of terms that have already been explored, starting as an empty list;
3. If the term is not a mutable pair, vector, promise, or non-empty list, it returns immediately because no other data structure could hold anything self-referential anyway.
4. If the term is one of the aforementioned types, `explore` checks if it's already on the list of used terms. If so, it adds the term to the list from step 1 and returns.
5. If the term is not already in the list of used terms, it recursively calls itself on every term in the list/vector/pair/promise, passing itself a version of the list of used terms that has the current term appended to it.

By passing the list of used terms to itself instead of maintaining it as a global variable, `explore` guarantees that if it runs into a term that's already on the list, that's because it's in a loop and not just because the original expression includes the same term twice in different places.

Using the list from `mapcycle` and a special object that keeps track of what numbers have been assigned to self-referential terms, `scmprint` deals with data structures by printing their essential elements (such as parentheses and whitespace for lists) directly and recursively calling itself to print each individual term. It also accepts as an optional argument a function to direct its output (defaulting to `cat`), so that it can easily be repurposed to write expressions to files or build them as strings. When it encounters objects of types it's unfamiliar with, its default behavior is to use `capture.output` to save the result of passing them to R's `print` function, then pass the resulting text line-by-line to the specified output function.

3. Environments

An environment is basically just a collection of variable bindings. Environments are hierarchical, with any variables not bound in a given environment having whatever values they had in its parent, or its parent's parent, and so on. Environments in my version of Scheme are implemented as pseudo-objects, made possible by the fact that R includes both first-class functions and lists with named items. When the function `extend_env` is called to create a new environment, it creates a list to hold the new environment's bindings, as well as methods to define, reassign, and look up values in the new environment, passing requests to the parent environment as necessary. It then creates a list of those methods by name, sets its class to `scmenv`, and returns it.

Scheme is unlike most other languages in that it has two separate commands for binding variables: `define` always binds the variable in the environment it was called from, whether that environment already contained a variable by that name or not, and `set!` seeks out and overwrites the existing binding of a variable, whether that's in the environment it was called from, or in the parent of that environment, and so on, throwing an error if it can't find a binding to overwrite. The exclamation mark in `set!` is there because it can change things outside of its immediate environment, a trait it shares with the similarly named functions `set-car!`, `set-cdr!`, and `vector-set!`.

Within Scheme, environments can be accessed by calling the following zero-argument functions:

- `global-env` returns the Scheme global environment, which can also be accessed as a variable under the name `user-interaction-environment`;
- `reference-env` returns a copy of the Scheme global environment as it was when the interpreter first launched, with only the predefined bindings;
- `null-env` returns an environment with no bindings whatsoever, though it will still be able to read the bindings in the global R environment;
- `current-env`, which is technically a special syntactic form but works like a function, returns the environment in which it was called.

As an example of the difference, if the variable `x` has the value `7` in the global Scheme environment, then the expression `(let ((x 2)) (eval '(+ x 3)(global-env)))` will return `10` and the expression `(let ((x 2)) (eval '(+ x 3)(current-env)))` will return `5`.

3.1 Eval and Apply

Scheme, like pretty much every version of Lisp since 1962 (McCarthy et al. 1962), is based around the two interconnected functions `eval` and `apply`, which Smalltalk developer Alan Kay once described as "Maxwell's Equations of Software" (Feldman 2004).

Of the two functions, `apply` is the simpler one to explain. It performs a function call of the function it receives as its first parameter by using the second parameter as the list of arguments for the function. For this, when passed a procedure and a list of arguments, it extends the procedure's environment by binding the variables in the procedure's parameter list to the corresponding values in the list it was passed. This is also why it was so important to retain support for improper lists, as they allow procedures to accept variable numbers of arguments. For example, if the procedure `f` has the parameter list `(a b . c)`, then calling `apply` on `f` and the list `(1 2 3 4 5)`, which can also be written as `(1 2 . (3 4 5))`, would create an environment in which `a` was bound to `1`, `b` was bound to `2`, and `c` was bound to the list `(3 4 5)`. Once this new environment has been created, `apply` then uses `eval` to evaluate the terms in the procedure's body one by one in this new environment.

The other half of the pair, `eval`, forces evaluation to be performed on the arguments. For this, it takes as its arguments an expression and an environment, then evaluates the expression according to the following rules:

- If the expression is a symbol, `eval` returns the value bound to it in the specified environment.
- If the expression is neither a symbol nor a list, `eval` returns it as-is. For instance, the number `23` evaluates to `23`, and the string `"skidoo"` evaluates to `"skidoo"`.

- If the expression is a list, its first term is a symbol, and that symbol is one of Scheme's keywords (`if`, `begin`, `quote`, etc), it's evaluated by the rule corresponding to that symbol. For example, the rule for an expression starting with `quote` is to return its second term verbatim, so `(quote (+ 2 2))` evaluates to the list `(+ 2 2)`.
- If the expression is a list, but its first term either isn't a symbol or is a symbol that doesn't match a Scheme keyword, `eval` recursively calls itself on every term in the list one by one then calls `apply`, with the result of evaluating the first term as its procedure and the results of evaluating the rest of the terms as its argument list. This means that a function with no arguments is called simply by surrounding its name in parentheses; the expression `runtime` will return the function `runtime`, but the expression `(runtime)` will call the function `runtime` and return the result. This also means excess parentheses are important in a way they wouldn't be in other languages; while R has no problem evaluating `(2)+2` to get `4`, trying to evaluate `(+ (2) 2)` in Scheme will throw an exception because `2` isn't a function.

My particular implementation is somewhat atypical because in addition to a Scheme procedure, my version of `apply` will accept an R function or a string as its first argument, passing them to the function `do.call`, which is basically R's equivalent of `apply`. Since `eval` handles any list whose first term isn't a Scheme keyword by passing it to `apply`, this allows users to call R functions from within Scheme.

Furthermore, in cases where an R function shares the name of a Scheme procedure, users can call the R version directly by using a string. For example, in Scheme, the symbol `lcm` is bound to a procedure that takes an arbitrary quantity of numbers as its arguments and returns their least common multiple, while the R function `lcm` takes a single number as its argument, which it converts to a string ending in `"cm"`. Thus, the Scheme expression `(lcm 28 49)` will evaluate to the number `196`, and the expression `("lcm" 28)` will evaluate to the string `"28 cm"`. R functions whose names aren't shadowed, such as `cos`, can be called either way, though there's a slight difference in how the two cases are handled by `apply`: if the function it's passed is an actual R function, it will call `do.call` with the parameter `quote` set to true, which will allow symbols to be passed to functions as arguments. However, because this can result in unpredictable behavior for some functions, calling an R function as a string will cause `do.call` to be called with `quote` set to false. As an example, `(typeof (quote x))` will return the string `"symbol"`, whereas `("typeof" (quote x))` will either return the type of whatever variable is bound to `x` in the global R environment or throw an error if it's not bound at all.

4. Tail Call Optimization

The basic idea behind tail call optimization is that if all function `f` has left to do is call function `g` and return the result, a combination of actions known as a tail call, then instead of making a new stack frame for `g` and adding it to the top of the stack, it's more efficient to replace the stack frame for `f` with one for `g` and let it directly return its value to whatever called `f`, since all `f` was going to do was pass it on verbatim anyway (Sebesta 2012). The most common case, called tail recursion, is when `f` and `g` are the same function, but proper tail call optimization works just as well with functions calling each other in tail position.

Tail call optimization has been an essential feature of Scheme since its inception (Sussman and Steele 1975). In addition to allowing more direct representations of recursive functions, it also allows the kind of looping that in most languages would require special syntactic forms like `for` or `while` to be accomplished through regular function calls (Abelson, Sussman, and Sussman 1996). In fact, the `do` keyword in Scheme is basically just shorthand for defining and calling a simple tail-recursive function.

4.1 Simple Tail Recursion Elimination

The simplest type of tail call is one that a function makes to itself. This can almost always be eliminated by placing the body of the function inside a loop and replacing the tail call with a set of commands to reassign the function's variables. This is an approach which is commonly used even in languages that don't support full tail call optimization (Goodrich, Tamassia, and Goldwasser 2014). This approach was used by Peter Norvig in his second Python-based Lisp interpreter (Norvig 2010b), but it's not without some rather severe downsides. Because everything has to remain with a single infinite loop, Norvig's version of `eval` can't call or be called by other functions (or rather, those calls won't benefit from the optimization). This has two major effects: first, it's impossible to define functions that evaluate specific syntactic forms and have the main `eval` function dispatch to them dynamically. Thus, adding a new form means directly editing the `eval` function, with the attendant risks of potentially breaking some part of it that was already working. Second, and perhaps more importantly, `eval` and `apply` can't call each other; function application is handled directly by setting the environment and expression variables and letting the next loop process them. If we wanted to make `apply` available as a function within Norvig's Lisp, we would need to define it separately, and any future changes to how function application worked would need to be made in both that function and within `eval` itself.

4.2 The Trampoline Pattern

Another approach to implementing tail call optimization in languages that don't natively support it is the trampoline pattern, which works by rewriting recursive functions so that instead of calling themselves directly, they return functions that call them when used, essentially improvising a form of lazy evaluation (Michel 2018). These functions are then passed to a function that basically consists solely of a `while` loop that repeatedly overwrites the function's argument with the value returned by calling it, stopping only when the result is something other than a function. This works on both recursive and non-recursive tail calls, but the extra overhead of each call having to create its own function makes it slow.

4.3 The Analyzing Interpreter

Fortunately, I had a solution. The analyzing interpreter described in section 4.1.7 of *Structure and Interpretation of Computer Programs* (Abelson, Sussman, and Sussman 1996) implements a form of just-in-time compilation by converting Scheme expressions into functions that each take an environment as their only argument and, when called, evaluate the expressions they were created from in that environment. The analysis process also only creates the function after it's done as much evaluation as possible without knowing the environment in which they're going to be run. For example, the function to analyze `if` statements starts by analyzing the predicate, consequent, and alternative clauses, then creates a function that, when called, calls the evaluated form of the predicate clause on the environment it was given, then uses the result to decide whether to call the analyzed form of the consequent or alternative clause.

This means that calling the function does considerably less work than evaluating the expression from scratch would have. And since the analysis is only done once per expression, the savings become more pronounced the more times the same function is called, which means it's particularly useful for iteration by tail-recursive functions. Thus, we have a way to convert expressions into functions that not only doesn't slow everything down, but in fact improves performance.

Another benefit of this approach is that since all analysis finishes before any evaluation begins, the `analyze` function doesn't have to recurse along with the expressions it's evaluating. Consider the following simple Scheme code, which I used during development to make sure my interpreter implemented tail call optimization properly:

```
(define (iter n)
  (if (< n 0)
      'done
      (iter (- n 1))))
(iter 9999)
```

When processing the definition, `eval` would create a function whose body was the `(if ...)` statement above and whose environment was the environment in which `define` was called (in this case, the global environment). When evaluating the call to `(iter 9999)`, a version of `eval` that wasn't specifically written to allow tail calls would first evaluate the expression `iter`, determine that it was a procedure, evaluate its arguments one-by-one (in this case, the only one is `9999`), and then call `apply` with the procedure and its list of evaluated arguments. Next, `apply` would create an environment whose parent was the environment attached to `iter`, in which `n` is bound to the number `9999`. It would then ask `eval` to evaluate the `(if ...)` expression in that environment, which would result in `eval` calling `apply` again, and so on. Obviously, these repeated calls would eat up stack space, resulting in an error, which is why `eval` would have to be written in a more convoluted fashion.

Now consider how an analyzing interpreter would handle the same set of expressions. First, it would analyze the `(define ...)` expression and create a function that, when called with an environment, would pass it to the analyzed form of the procedure body and bind the result to the name `iter` in that environment. All of this, however it was done, would finish before any calls to `iter` were made. When evaluating the expression `(iter 9999)`, all `analyze` would do is notice that `iter` is not a syntactic keyword, which means the expression should be treated as a function application. It would then create a function that, when passed an environment, looks up the value of `iter` in that environment and passes the result, along with a list consisting of the number `9999`, to `apply`, which then turns to the trampoline for actual execution. At that point, no further calls to `analyze` would be made, meaning that the total number of calls, and thus the potential maximum size of the stack, is completely unaffected by whatever value is passed to `iter`. Thus, `analyze` can be implemented without worrying about running out of stack space. While it's theoretically possible for sufficiently deeply nested code to cause a stack overflow with `analyze`, you would pretty much have to be doing it on purpose, and it would probably be better practice to make the most deeply nested parts into their own procedures anyway.

In short, this approach allows me to implement the core of my interpreter in whatever manner I see fit while still reaping the benefits of tail call optimization. The approach I ended up using was a data-directed one: when passed an expression that takes the form of a list whose first term is a symbol, `analyze` will check a table of analysis functions to see if there's one registered to that symbol and, if so, pass the expression to it. If not, it will default to treating the expression as a function application, creating a function that, when passed an environment, will evaluate each term in that environment and pass the results to `apply`, with the first term as the procedure and the remaining terms as its arguments. Thus, to add a new syntactic keyword, one needs only write an analysis function for it and add it to the table, all without modifying any existing code.

4.4 The Virtual Machine

The functions created by the analyzing interpreter differ from those used in the standard trampoline pattern in two major ways: first, because functions are a valid type of return value in Scheme, simply testing whether or not a given value is a function is insufficient to tell whether or not it needs to be run again. To fix that, all functions created by `analyze` and its various sub-functions are given the class `thunk`. It is named after an observation made during the development of ALGOL 60 that "most of the analysis of ('thinking about') the expression could be done at compile time; thus, at run time, the expression would already have been 'thunk' about" (Abelson, Sussman, and Sussman 1996). Rather than checking whether or not its variable is a function, each iteration of the `while` loop checks whether or not its class matches. Second, and more difficult to address, each thunk takes one argument rather than zero.

To explain why this was an issue, it helps to take a closer look at the changes I had to make to the optimizing interpreter in order to keep it compatible with the trampoline pattern. The first two are fairly straightforward reflections of the way the pattern works. First, whenever a thunk would normally call another thunk in tail position, it returns it instead and allows the trampoline function (which has the name `run()` in my interpreter) to make the actual call. Second, because there's no guarantee that any given thunk will immediately produce a return value, non-tail calls to other thunks also have to go through the trampoline. That is, to evaluate the code represented by the thunk `proc` in environment `env`, you would need to call `run(proc,env)` instead of just calling `proc(env)`.

The other side of the coin, and the one that required me to create a virtual machine, was the way the trampoline pattern itself had to be adjusted to work with the fact that each thunk required an environment in which to run. If a thunk wants to make a tail call to another thunk that will be executed in a different environment, it can't simply return the thunk and let `run` do the rest, and it can't call `run` itself because that would add another frame to the call stack, defeating the purpose of using the trampoline in the first place. The solution I came up with was to implement a virtual machine.

The virtual machine used by my Scheme interpreter is essentially an extremely simple, extremely abstract computer. Thunks are analogous to instructions, directly invoking the `run` function is equivalent to using the `call` instruction. Environments are equivalent to stack frames, and the `while` loop inside the `run` function is the equivalent of a fetch-decode-execute loop. As mentioned above, executing a tail call is simply a matter of overwriting the current stack frame if necessary and jumping to the correct instruction. In this case, that means overwriting the environment used by the currently active instance of `run` and returning the next thunk.

Implementation-wise, the virtual machine is an object consisting of a stack, the function `setenv`, and a modified version of the `run` function. When called, the first thing the modified `run` does is push the current `setenv` function onto the stack, then replace it with one that, when called, changes the environment that it (the currently active instance of `run`) uses. It then does the usual trampoline thing of repeatedly overwriting the variable `proc` with the return value of the call `proc(env)`. However, this is enclosed in a call to `tryCatch`, with the command to restore the previous `setenv` function from the stack afterward as a `finally` term to guarantee that the stack is restored even if errors are encountered during evaluation.

Similarly, because it's frequently called during evaluation, `apply` needed to be reworked for trampoline support. In addition to a procedure and a list of values, it also takes a third parameter, `active`, which has a default value of false. If `active` is set to true, calling `apply` on a Scheme procedure will create a new environment and call `run` on that environment and the body of the procedure. If `active` is false, which is the default so that it works properly when called from within Scheme, it will instead call `setenv` on the newly extended environment and return the body of the procedure. In either case, it will still respond to being passed an R function of a string by calling `do.call`, because R functions don't work with the trampoline directly anyway.

5. The REPL

While the interpreter includes functions for executing Scheme code directly within R, its primary interface is the read-eval-print loop, or REPL, which can be started by calling the function `repl`. When called, the first thing `repl` does is set the global option `warn` to 1, which causes warning messages to print immediately rather than waiting for the top-level function (which, in this case, is `repl` itself) to finish first. This way, warnings generated by the user's actions will be shown in a timely fashion rather than remaining invisible until the REPL is exited. Next, `repl` uses R's `callCC` function to capture a continuation that will exit the function, which it then uses to implement a function that will, when called, restore the `warn` option to its original value and use the previously captured continuation to cause `repl` to immediately exit and return the value that was passed to it. It then binds this function to the name `quit` in the global Scheme environment, thus creating a way for users to exit the REPL.

With the setup out of the way, `repl` then enters the loop proper. The first thing it does is output the prompt `";;; RScheme input:"`, then call its input function. By default, this is a function that uses R's `readline` function to get input directly from the user, allowing multi-line input by continuing to call `readline` until the total number of opening and closing parentheses the user has entered match. Once that happens, it passes the result to the lexer and parser, which convert it into a list of Scheme expressions. This is done within a call to `tryCatch`, enabling `repl` to detect any errors thrown by the lexer or parser, relay them to the user, and prompt for better input.

Once it's acquired input that can be parsed without triggering a syntax error, `repl` outputs the prompt `";;; RScheme output:"` and evaluates the expression(s) in the list it got from the parser. Each evaluation is, again, done within a call to `tryCatch` so that `repl` can respond to errors by displaying them and continuing to run, rather than by immediately exiting. After evaluating each expression, it checks whether the result is either null or the special symbol `#<void>`. If not, it outputs the result and moves on to the next term. Once all terms have been evaluated and printed, it goes back to the top of the loop, outputting the `";;; RScheme input:"` prompt and waiting for more user input.

6. Scheme Commands

By default, my Scheme interpreter includes 21 syntactic forms and over 150 predefined functions. While going through all of them would be somewhat excessive, the following should be enough to serve as a representative sample.

6.1 Essential Syntax

While my interpreter includes code to handle all syntactic forms directly, which I did partly for efficiency and partly because I felt I would learn more from implementing them that way, it’s entirely possible to define most of them in terms of each other, as will be shown with `let` later in this paper. The only forms that actually need to be derived directly are `begin`, `define`, `set!`, `lambda`, `quote`, `define-macro`, `current-env`, and either `if` or `cond` (either can be defined in terms of the other, but at least one of them needs to be written directly). I already discussed `define`, `set!`, and `current-env` in the section on environments, and `lambda` and `define-macro` will have their own sections below, but this is as good a place as any for a brief overview of `begin` and `if`.

Scheme uses `if` to handle simple conditional statements. The expression is always in the form of either `(if test consequent)` or `(if test consequent alternative)`. In either case, the `if` statement starts by evaluating `test`, then decides what to do next. If the result of evaluating `test` is true, it evaluates `consequent` and returns the result. If not, it either evaluates `alternative` and returns the result or, if `alternative` is missing, returns the placeholder symbol `#<void>`. Notably, `test`, `consequent`, and `alternative` are each single expressions; to create conditional blocks that do multiple things, the `begin` statement is needed.

The way `begin` works is extremely simple: a `begin` expression consists of the symbol `begin` followed by any number of Scheme expressions, all of which are evaluated in order, with the value of the last one also being the value of the `begin` expression itself. This allows any arbitrary sequence of expressions to be treated as a single item, which is useful in forms like `if` and `delay` that take a limited number of parameters. In some variants of Scheme, `begin` can in fact be defined in terms of other commands. In fact, the section 7.3 or the R5RS specification gives two different ways to do exactly that (Kelsey, Clinger, and Rees 1998). However, those implementations differ from mine in that they don’t allow `define` commands inside of `begin` statements, which I chose to allow as a way that users could manually override the way my interpreter handles definitions within functions, which will be elaborated upon in the section below about variable bindings.

6.2 Arithmetic Operators

While I could theoretically have just used the arithmetic operators defined in R, the Scheme versions are able to handle arbitrary numbers of arguments, at least for the operators where either the transitive or associative property holds. For example, while converting `(+ 1 2 3)` to `1+2+3` and `(< 1 2 3)` to `(1<2)&(2<3)` is perfectly intuitive, the same can’t be done with the expression `(!= 1 2 2)` because it’s not clear whether it would map to `(1!=2)&(2!=2)`, which is false, or `!((1==2)&(2==2))`, which is true. Besides, I wanted to extend the way they work with R’s numeric vectors to apply to lists. For example, the Scheme expression `(+ 10 '(1 2 3))` evaluates to the list `(11 12 13)` in my interpreter, much like the R expression `10+(1:3)`. I did the same thing with `sqrt`, which I had to modify anyway so that being passed a negative number would cause it to return a complex number rather than throwing an error, and with `atan`, which I had to modify so that if passed two values `y` and `x`, it would find the angle of the line passing through the point (x,y).

6.3 Higher-order functions

While arithmetic operators are implemented to automatically handle lists, other functions are not. However, Scheme already includes a way to handle this. The procedure `map` takes a function as its first argument, followed by one or more lists, and returns a list of the result of applying the function to the first term in each list, then to the second, and so on. For example, the Scheme expression `(map abs '(1 -2 3 -4))` would evaluate to the list `(1 2 3 4)`. Similarly, the functions `andmap` and `ormap` also take a function and a list or lists to iterate through, but instead of a list, they return a boolean value; `andmap` iterates through the list(s) until it either gets a false value, in which case it returns false, or runs out of terms, in which case it returns true. Likewise, `ormap` iterates until it gets a true value, in which case it returns true, or runs out of terms, in which case it returns false. For example, `(andmap < '(1 4) '(2 3))` is false because 4 is not less than 3, and `(ormap < '(1 4) "(2 3))` is true because 1 is less than 2.

Other higher-order functions include `for-each`, which executes the same way as `map` but doesn't return anything, `filter`, which takes a function and a list and returns only the values from the list for which the function didn't return false, and `fold-left` and `fold-right`, which each take a two-argument function, initial value, and list of terms, and repeatedly replace the initial value with the result of calling the function with it and one term from the list. The difference is that `fold-left` uses the initial value as the function's first argument and goes through the list from left to right, while `fold-right` uses the initial value as the function's second argument and goes through the list from right to left. For example, `(fold-right / 1 '(2 3 4))` evaluates to `2/(3/(4/1))`, and `(fold-left / 1 '(2 3 4))` evaluates to `((1/2)/3)/4`.

6.4 Lambda expressions

When using higher-order functions, there will be times when a function is needed only once. For example, suppose you have a large list of numbers and want to collect all the values from it that are between 5 and 15. You could define a function `between-5-15` and use it with `filter`, but this seems somewhat wasteful as it's unlikely that a function for testing whether or not a number is between 5 and 15 is something you'll be reusing. For this type of situations, Scheme provides `lambda` expressions, also sometimes known as anonymous functions. If we want a collection of all the items in the list `nums` that are between 5 and 15, one approach would be to use the command `(filter (lambda (n)(< 5 n 15)) nums)`. The interpreter also internally constructs lambda expressions when dealing with function definitions; `(define (f x) (+ x 3))` is essentially the same as `(define f (lambda (x) (+ x 3)))`, for example. It's also possible to call `lambda` expressions directly; `((lambda (x y)(+ (* 2 x) y)) 5 7)`, for example, will evaluate to `31`, though this is obviously less common as users could more easily enter the expression `(+ (* 2 5) 7)` in the first place. The one advantage of using a `lambda` expression in that context would be to bind the numbers to specific variables, but that's where Scheme's variable-binding forms come in.

6.4 Variable Binding

In practice, Scheme programmers will typically use some variety of `let` statement to bind variables instead of defining a `lambda` expression directly, though the two generally behave identically. In fact, in some versions of Scheme, the evaluator would internally convert the expression `(let ((x 5)(y 7)) (+ (* 2 x) y))` into `((lambda (x y)(+ (* 2 x) y)) 5 7)` before evaluating it (Abelson, Sussman, and Sussman 1996). Not counting the “named `let`” form, which is more of a looping construct and will be discussed later, there are three varieties of `let` statements defined in the R5RS standard (Kelsey, Clinger, and Rees 1998), and a fourth, `letrec*`, was added by R6RS (Sperber et al. 2007). All of them are implemented in my version of Scheme.

The most basic variable-binding form, `let`, evaluates the variables it’s binding simultaneously in its existing environment. For example, if the following code

```
(let ((a 2)
      (b (+ a 5)))
  (* b 6))
```

were evaluated in an environment in which the variable `a` has the value `5`, the result would be `60`, not `42`. This is because the expression `(+ a 5)` is evaluated in the environment in which the `let` statement was made, where `a` has the value `5`. The second form, `let*`, behaves more intuitively by evaluating its variables sequentially: while it evaluates the value of its first variable in its own environment, it evaluates its second variable in an environment in which the binding for the first is already in place. Were there a third variable, it would be evaluated in an environment in which the first two were already bound, and so on. If the above code started with `let*` instead of `let`, it would indeed evaluate to `42`.

The third form, `letrec`, evaluates its variables simultaneously just like `let`, but it evaluates them in the environment in which they’ll be defined. This enables it to bind functions that can call themselves or each other. It also doesn’t allow variables to reference each other while being defined (functions don’t count, as they’re not actually evaluated until they’re called); if the code above started with `letrec`, it would throw an error because the expression `(+ a 5)` was being evaluated in an environment in which the value for `a` had not yet been determined.

The final form, `letrec*`, attempts to combine the best of both worlds: variables are evaluated in the environment in which they’ll be bound, allowing for recursive functions, but they’re bound sequentially, so each variable can reference its predecessors. The code above would behave exactly the same if it started with `letrec*` as if it started with `let*`, but the following:

```
(letrec* ((b (+ a 5))
          (a 2))
  (* b 6))
```

would throw an error, as `b` was trying to reference a definition of `a` that hadn’t been evaluated yet. By contrast, if the above started with `let*` (or, for that matter, `let`), it would simply evaluate to 60, as `b` would use the “old” binding for `a`.

Inside of definitions, `lambda` expressions, and `let` blocks, uses of the `define` keyword are scanned out and treated as parts of an implicit `letrec*` form in which the rest of the expression will be evaluated. The standard approach is to treat them like bindings in a `letrec` block, but I feel processing them sequentially is a better idea, as following the line `(define a 2)` with `(define b (+ a 5))` would naturally use the correct binding for `a` to set `b` to `7`. At the same time, if the definitions are in the opposite order, it's better to throw an error because `b` should be assumed to be trying to use the definition of `a` in its own environment, which hasn't yet been executed, rather than whatever value `a` may have had outside the current context. If users want their `define` commands to be evaluated purely sequentially for some reason, they also have the option of placing them inside a `begin` statement, as the function that scans and isolates definitions will leave the contents of `begin` blocks alone.

6.5 Looping Constructs

While, as noted above, the existence of tail call optimization means that Scheme doesn't actually need special structures to control program flow the way most languages do, it still has two of them. The one most similar to the types found in other languages is `do`, which is essentially a `for` loop that has an arbitrary number of control variables and a specific set of instructions to go through once it finishes executing. It takes the form

```
(do ((var1 init1 next1)
     (var2 init2 next2))
  (test end1 end2)
  alt1
  alt2)
```

where the `var` terms are the names of the variables, the `init` terms are their initial values, and the `next` terms are evaluated at the end of each loop to determine what values their variables have the next time around. The loop starts by creating a new environment in which each `var` name is bound to the result of evaluating the corresponding `init` term, evaluating the `test` term in that environment. If it's true, the `end` terms are evaluated one by one, with the result of the last one being the return value of the expression. If `test` is false, the `alt` terms are evaluated one by one, after which the `next` terms are evaluated, the results are bound to the corresponding `var` names in a new environment, and the loop starts over, evaluating the `test` expression in this new environment. There can be any number of `var`, `init`, `next`, `end`, and `alt` terms; I simply limited the above to two of each to save space.

The more general form, known as the named `let`, is a variant of the standard `let` expression. Syntactically, it takes the form

```
(let name ((var1 val1)
           (var2 val2))
  body1
  body2)
```

The execution is also simple: a function that takes the `var` terms as parameters and executes the `body` terms in order is bound to `name`, then called with the `val` terms as its arguments. The `body` terms can call `name` just like any other function, allowing for both a more varied iterative structure than `do` (because different parts can call `name` with different values) and also for non-tail recursion.

6.6 Delayed Evaluation

The `delay` command takes any other Scheme expression and returns an object that, when passed to `force`, will evaluate that expression and return the result. The result will also be stored, so subsequent calls to `force` with the same item will give the same result, but won't actually run again. Take the following as an example:

```
(define x (delay (begin (display "Hello!") 29))
(force x)
(force x)
```

The first `(force x)` will display the string `"Hello!"`, then return the number 29. The second `(force x)` will return the number 29, but it won't display anything because it's just accessing the cached result from the first time rather than running it again. The difference is also visible if `x` is printed; before being forced, it prints as `#<promise:(begin (display "Hello!") 29)>`, and afterward it prints as `#<promise!29>`.

6.7 Input and Output

My Scheme interpreter includes basic file I/O through simulated "ports", usable with the `read`, `read-char`, `peek-char`, `display`, `write`, and `newline` commands. The input commands can be called with either no arguments, in which case they'll read input directly from the keyboard, or one argument, which must be an input port (actually implemented as a copy of the lexer that has been fed the contents of the file). Both `read-char` and `peek-char` read a single character of input, though `read-char` removes said character from the port while `peek-char` leaves it in place, and `read` reads a single Scheme expression (assembled with a call to `lex.next`). Input ports can be created in two ways: the Scheme function `open-input-port` takes a string as its argument and returns an input port for the contents of the file whose name matches that string, and the function `call-with-input-port` takes two arguments, the first of which is a string and the second of which is a function or procedure, which will be called with an input port matching the specified file as its only argument.

Output functions are similar; `newline` takes either no arguments or one argument, and `display` and `write` each take either one or two arguments. In either case, the last argument will be an output port (actually implemented as a function that appends whatever it's called with to the specified file), and omitting that argument will cause the function to display its output on-screen. The only difference between `display` and `write` is that when passed a single string to print, `display` will print it without quote marks and `write` will include them. Output ports can be created with either `open-output-port` or `call-with-output-port`, which work essentially the same way as their counterparts above. Optionally, `open-output-port` can be called with a second argument, which tells it whether or not to overwrite the file it opens.

6.8 Error Handling

The way my Scheme interpreter handles errors is the `with-handlers` special form, which is essentially a cross between the eponymous function in Racket (Flatt and PLT 2023a) and the `tryCatch` function in R. The syntax is

```
(with-handlers handlers
  term1
  term2)
```

where there are any number of `term` terms and `handlers` is a list of named items, containing some subset of `warning`, `error`, and `finally`. The values for `warning` and `error` are single-argument functions, and the value for `finally` is a single Scheme expression. The `term` terms are executed in order, with the value of the last also being the value of the `with-handlers` expression. Should any warnings or errors arise in the process, the corresponding functions in the `handlers` list will be called if available, with their argument being the exception thrown by the system, and whatever the handler returns will become the value of the `with-handlers` expression. Either way, the `finally` term, if one is given, will be evaluated after everything else, but its value will not be returned.

As an example, the following will display the message `"You were warned"` and return the number `7` if anything in *body* generates warnings, and it will display the message `"Something went wrong"` and return the number `-19` if anything in *body* generates an error. Either way, it will display the message `"This is inevitable"` before returning.

```
(with-handlers (#:warning (lambda (w)
                            (display "You were warned")
                            (newline)
                            7)
                #:error (lambda (e)
                          (display "Something went wrong")
                          (newline)
                          -19)
                #:finally (display "This is inevitable"))
  body)
```

If the `error` or `warning` terms are missing, the things they're meant to handle will be treated as though they weren't in a `with-handlers` block at all: warnings will be displayed as they arise but won't stop evaluation, and errors will stop evaluation and display the corresponding error message.

7. Interfacing with R

Because `apply` is implemented as an R function (though it's renamed to `scm.apply` because the name `apply` was already in use by an unrelated R function), it can be used along with R's `...` operator to create R functions that call arbitrary Scheme procedures, using the format

```
function(...) {
  scm.apply(procedure,list(...),TRUE)
}
```

where *`procedure`* is the Scheme procedure we want to call. This is the basis of the command `export` in my version of Scheme, which takes the name of a variable, looks up its value in the current Scheme environment, creates a function to call it using the format above if the value is a procedure, and binds the result to the variable's name in the global R environment. For example, after executing the following two commands

```
(define (square x)(* x x))
(export square)
```

There will be a function called `square` in the global R environment, and calling `square(5)` in R will return the number `25`, as expected.

8. Macros

Another use for the ability to call Scheme procedures directly from R is the creation of macros. At the base level, a macro is a Scheme procedure whose arguments are passed to it without being evaluated and whose return value is a different Scheme expression. They are a useful feature that allows the user to do things like emulate reference parameters. This can be easily implemented by creating a function in the form

```
function(exp) {
  scm.apply(macro,exp[-1],TRUE)
}
```

which, when passed the expression `exp` will apply the procedure *`macro`* to it and return the result. My interpreter actually keeps a table of macros, each of which is a function defined in the above form, and calls them whenever it's passed an expression whose first term is a symbol that corresponds to an entry in the table. It actually uses a `while` loop to do this repeatedly, a process known as macro expansion, as it's entirely possible for macros to be defined in terms of other macros. For example, an implementation of Scheme could use a macro to change `do` expressions into `letrec` expressions, another macro to turn `letrec` expressions into `let` expressions, and another to turn `let` expression into `lambda` expressions as shown below.

If `let` wasn't already defined in the interpreter, the non-named form would be easy to implement as a macro, converting a `let` expression into one that uses `lambda` to create a one-off function whose arguments are the variables we want to bind, then calls it with the corresponding values:

```
(define-macro (let pairs . body)
  (cons (cons 'lambda
              (cons (map car pairs) body))
        (map cadr pairs)))
```

Using this macro, the expression `(let ((a 1)(b 2))(list a b (+ a b)))` would be converted to `((lambda (a b)(list a b (+ a b))) 1 2)`. In addition to `define-macro`, my implementation of Scheme includes the function `remove-macro`, which deletes macros that the user no longer wants. For example, if a user were to enter the above definition of `let` and then run into problems because it doesn't support named `let` statements, they could then use `(remove-macro 'let)` to remove it and revert to the interpreter's default handling of the `let` command. To help with debugging, I also included the function `test-macro`, which takes a Scheme expression as its argument and returns the result of applying a macro to it if applicable. For example, I called `(test-macro '(let ((a 1)(b 2))(list a b (+ a b))))` to get the result of using the `let` macro at the start of this paragraph. Note that since they're both functions, `remove-macro` and `test-macro` need their arguments to be quoted.

8.1 Quasiquotation

While macros that put together new expressions using commands like `cons` and `list` are perfectly functional, they aren't always intuitive. Fortunately, Scheme has a solution in the form of quasiquotation. As the name implies, quasiquotation is similar to quotation. The main difference is that within a `quasiquote` expression, any term starting with `unquote` is evaluated normally. For example, if `x` has a value of `7`, then `(quasiquote (x (unquote x)))` will evaluate to the list `(x 7)`. There is also an `unquote-splicing` term which is similar, except that its value must evaluate to a list, which will be spliced into the expression instead of left as-is. For example, if `y` is bound to the list `(1 2 3)`, then `(quasiquote (x (unquote y) z))` will evaluate to the list `(x (1 2 3) z)`, but `(quasiquote (x (unquote-splicing y) z))` will evaluate to the list `(x 1 2 3 z)`. Each term also has its own reader macro: the backtick `` ` `` for `quasiquote`, the comma for `unquote`, and the symbol `,@` for `unquote-splicing`. Thus, if `a` is bound to the number `1` and `b` is bound to the list `(2 3 4)`, the expression `` `(a ,a ,b ,@b b) `` will be equivalent to `(quasiquote a (unquote a)(unquote b)(unquote-splicing b) b)`, which will evaluate to `(a 1 (2 3 4) 2 3 4 b)`.

Naturally, the "fill in the blank" nature of quasiquotation is particularly useful for macros. For example, using quasiquotation, the macro for `let` above can be redone in a much more intuitive fashion:

```
(define-macro (let pairs . body)
  `((lambda ,(map car pairs)
      ,@body)
    ,@(map cadr pairs)))
```

8.2 Macro Hygiene

One potential issue with the simple setup provided by `define-macro` is that if a macro uses any terms that weren't in the expression passed to it, they can interfere with existing code. For example, take the following macro, which implements C-style `for` loops using named `let`:

```
(define-macro (for head . body)
  (let ((var (car head))
        (test (cadr head))
        (next (caddr head)))
    `(let loop (,var)
       (if ,test
           (begin
            ,@body
            (loop ,next))))))
```

If the name `loop` is already used at some point within the body for something else, this will cause errors as the interpreter has no way to know which uses of `loop` refer to that and which were added by the macro. More modern versions of Scheme eschew the use of `define-macro` entirely in favor of the special forms `syntax-rules` (Kelsey, Clinger, and Rees 1998) and `syntax-case` (Sperber et al. 2007), which handle this issue automatically at the cost of being more complex to implement. There is, however, a simple fix that works with `define-macro` to solve this issue: the `gensym` procedure (Sitaram 1998).

In simple terms, `gensym` takes a symbol and returns a new symbol that, while similar, hasn't been used anywhere else in the program. Some implementations do this by checking the table Scheme uses to keep track of all existing symbols, but my interpreter uses R's native symbol type, and I don't know a user-accessible way to check R's symbol table. Instead, my version of `gensym` converts the symbol it's passed into a string, appends to it the string format of the current time in milliseconds, then appends a fifteen-digit random number given by R's `runif` function. In order for `gensym` to return the same symbol twice, it would have to be execute twice within the same millisecond, and both would have to receive the same set of digits from `runif`, out of the 10^{15} equally likely possibilities. Using `gensym`, the above macro could be rewritten as

```
(define-macro (for head . body)
  (let ((var (car head))
        (test (cadr head))
        (next (caddr head))
        (loop (gensym 'loop)))
    `(let ,loop (,var)
       (if ,test
           (begin
            ,@body
            (,loop ,next))))))
```

This would replace any uses of `loop` added by the macro with something like `loop168270059045630564593876595608 9`, while leaving any that already existed unchanged. And since `gensym` never returns the same symbol twice, it also wouldn't interfere with any `loop` symbols in use by other macros that followed this same methodology, which is why I picked this method over simply having it return a symbol like `#;loop` that the user couldn't have entered directly.

8.3 Reader Macros

The table of reader macros can be accessed with the `add.readermacro` and `rm.readermacro` functions in R, or the `add-reader-macro` and `remove-reader-macro` functions in Scheme. Combined with `define-macro`, this can allow for some major customization to Scheme's syntax, such as implementing the prefix form of the `++` operator from Java and C++, which can be done as follows:

```
(define-macro (inc! x)
  `(set! ,x (+ ,x 1)))
(add-reader-macro '++ 'inc!)
(define x 0)
(list ++x ++x ++x) ;returns (1 2 3)
```

Note that since `add-reader-macro` and `remove-reader-macro` are functions rather than syntactic forms, any symbols passed to them have to be quoted.

9. Conclusion

In this project, I created a Scheme interpreter that runs within R and is capable of both calling R functions directly and exporting Scheme procedures into functions that can be called from within R. The most educational part was probably the work I did to implement tail call optimization, both in figuring out that the functions generated by the analyzing interpreter could be used with the trampoline pattern and in the creation of the virtual machine to manage environments as stack frames. The main use for my project is to enable the creation of programs that benefit from both R's massive library of third-party utilities and Scheme's simple and extensible syntax.

Works Cited

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass. : New York: MIT Press ; McGraw-Hill.

Feldman, Stuart. 2004. "A Conversation with Alan Kay: Big Talk with the Creator of Smalltalk - and Much More." *Queue* 2 (9): 2030. https://doi.org/10.1145/1039511.1039523.

Flatt, Matthew, and PLT. 2023a. "10.2 Exceptions." https://docs.racket-lang.org/reference/exns.html.

———. 2023b. "4.11 Mutable Pairs and Lists." https://docs.racket-lang.org/reference/mpairs.html.

Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. Sixth edition. Hoboken, NJ: Wiley.

Kelsey, Richard, William Clinger, and Jonathan Rees. 1998. "Revised^5 Report on the Algorithmic Language Scheme." https://standards.scheme.org/corrected-r5rs/r5rs.html.

McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. 1962. *LISP 1.5 Programmer's Manual*. The MIT Press.

Michel, Thiery. 2018. "Understanding Recursion, Tail Call and Trampoline Optimizations." https://marmelab.com/blog/2018/02/12/understanding-recursion.html.

Norvig, Peter. 2010a. "(How to Write a (Lisp) Interpreter (in Python))." http://norvig.com/lispy.html.

———. 2010b. "(An ((Even Better) Lisp) Interpreter (in Python))." http://norvig.com/lispy2.html.

Peng, Roger D. 2022. *R Programming for Data Science*. https://bookdown.org/rdpeng/rprogdatascience/.

Schwarzer, Stefan. 2023. "Glossary of Racket Concepts." https://docs.racket-lang.org/racket-glossary/index.html#%28part._.Keyword%29.

Sebesta, Robert W. 2012. *Concepts of Programming Languages*. 10th ed. Boston: Pearson.

Sitaram, Dorai. 1998. "Teach Yourself Scheme in Fixnum Days." https://ds26gte.github.io/tyscheme/.

Sperber, Michael, R. Kent Dybvig, Matthew Flatt, and Anton von Straaten. 2007. "Revised^6 Report on the Algorithmic Language Scheme." https://standards.scheme.org/corrected-r6rs/r6rs.html.

Sussman, Gerald J., and Guy L. Steele. 1975. "SCHEME: An Interpreter for Extended Lambda Calculus," December. https://dspace.mit.edu/handle/1721.1/5794.