Thinking in Lists
An exploration of Scheme interpretation

Leiman, Oren oleiman@ucsc.edu
Levenberg, Ace llevenbe@ucsc.edu

March 18, 2013

Scheme is a minimal dialect of the programming language Lisp. It was designed by Gerald Sussman and Guy Steele with the goal of providing a small, mostly functional language core with robust tools for language extension. What follows is an exploration of the Scheme interpreter and its implementation in Python, Ruby, and Haskell. By comparing implementations in multiple languages, we aim to highlight not only the central aspects of the interpreter but also various differences between the three implementation languages. Python and Ruby are themselves quite similar, so we will primarily discuss these two as a unit in contrast to Haskell.

It should be noted that certain features of Scheme are missing from our implementations. To begin with, true Scheme specifies the inclusion of tail-call optimization, first-class continuations, and hygienic macros. We decided that these sorts of features would complicate our implementation in such a way as to obscure the essential simplicity and elegance that lies at the heart of Scheme. For similar reasons, our Haskell implementation does not do any sort of robust error handling. This paper is not about compiler-level black magic, but rather it is about the beauty and simplicity of a language specification which has stood the test of time despite its supposed lack of direct practical applications. We have included arithmetic, variable/function definitions, lambdas, list operations (but no true dotted lists), let expressions, boolean expressions, conditional statements, and a simple front end for experimentation.

The code for this project can be found at:

https://github.com/acelev/Hasche
Why these languages?

Scheme

Our main reasons for choosing Scheme are that it is small, that it is dynamically typed, and that the basic structure of its interpreter is both elegant and relatively simple. One reason for this is that it is homoiconic. This means that Scheme code itself is constructed of nested lists, which are also its primary data structure. In general, this makes Scheme particularly amenable to self-interpretation or modification. In our case, we were able to exploit this property by simply representing each program as a list in the implementation language. Combined with prefix notation for function application, this means that Scheme code essentially contains its own Abstract Syntax Tree.

S-expressions

S-expressions, short for “symbolic expressions”, are the structure which defines the Scheme AST and the output of the parse phase. S-expressions can be defined inductively as:

1. an atom, or
2. an expression of the form \((x.y)\) where \(x\) and \(y\) are s-expressions

That is, an s-expression is either an atom (i.e. a value or identifier) or a list of s-expressions. This is the brilliant thing about Scheme, and Lisp in general: the s-expression is the primary syntactic structure and has a very simple recursive definition, so each stage of both syntactic and semantic analysis essentially consist of deciding whether each element is an atom or a list!

Ruby/Python

This section is not really about comparing Python to Ruby, although they do differ on some fine points of style and semantics. The purpose of this was for us to each implement our own interpreter, ensuring that we both gained a thorough grasp of how the interpreter should work before moving on to Haskell.

Ruby and Python both include list-like structures which are flexible enough to store an entire Scheme program (i.e. a Scheme AST) without any modification of any kind. This is a huge boon and results in the possibility of a parser that simply emits a Scheme AST in the form of a Python/Ruby data structure (see Figure 1).

Another great feature of both languages is the dictionary/hash-table, whose use is encouraged and facilitated by the standard semantics of each language. These dictionaries are mutable, and they allow us to easily store state and pass it around the Read Eval Print Loop (REPL), updating and replicating it as needed. Okay, groovy, so what goes in the dictionary?
Now is a good time to mention that Ruby/Python dictionaries can be heterogeneous, just like their lists. What this means for us is that we can indiscriminately toss bindings into our global namespace without worrying about what the values are (numbers, strings, functions, etc.).

Both languages support first class closures (i.e. lambdas), so, making use of straightforward syntax for defining parameter lists at runtime, we were able to map Scheme lambdas directly to lambdas in the implementation language.

**Haskell**

The most obvious difference between Haskell and Python/Ruby is its large, static, extremely strict type system, which essentially strips us of all the slick conveniences outlined above. Although this is often considered a pain point of Haskell, we believe that, in the end, this produced code that is more elegant and easier to verify that its dynamic counterpart.

Haskell is purely functional, which means, among other things, that objects are not mutable. The consequence of this is that we lose our simple mechanism of updating and passing around state. Haskell has a way around this in the form of the State Monad, but this does more than to simply correct a supposed deficiency of functional languages; it actually buys us some interesting features and does wonders for readability.

This brings us to the huge advantage of using Haskell: Parsec. Parsec is a library for generating various types of parsers. In contrast to a traditional parser generator such as yacc/bison, which generate code in another language, Parsec parsers are first class Haskell objects. The lack of generated code (which is usually ugly boilerplate, and therefore impossible to debug directly) means that our parser code can be straightforward and integrated directly into the architecture of the interpreter.

**Dynamic vs Static Typing**

The dynamic type systems of Ruby and Python allow us to read Scheme ASTs directly into lists. Furthermore, the lack of types means that there is no need for high level abstractions to represent the structures of Scheme. The consequence of this is that the
parse phase can be extremely simple. Our parsers do essentially no syntax checking and literally translate strings representing Scheme programs into nested lists. This mapping is highlighted in Figure 1. The interpreter must now be responsible for both syntactic and semantic analysis of the code, since the parser really only buys us matched parentheses, which, since the code is the parse tree, is all that we need to discern the meaning of the program.

This process was a bit more involved in Haskell, but ultimately, by devising abstractions to represent Scheme forms, we were able to accomplish the same result. One property of Haskell lists is that they must be homogeneous with respect to some data type. Luckily, Haskell provides a rich syntax for defining algebraic data types, which are essentially composite types formed by combining other types[2]. In this way we can define SchVal, which contains a variety of constructors representing particular sorts of Scheme expressions. Up front, this introduces some design overhead, but the resulting lists of SchVals directly correspond to the Scheme ASTs mentioned above. Figure 2 shows our algebraic type, and the abstractions contained there provide a precise representation of the structure of Scheme.

### Methodology

Overall, our approach in practice differed significantly from that outlined in our proposal. Part of this arose from our initial ignorance about the process of writing an interpreter. For example, we proposed that the parser be the last thing implemented; however, we quickly found that testing and debugging our interpreter without a reliable parser was very tedious. At first, we thought that we could implement language features more or less sequentially, but in practice, it seems that one must take a more holistic view of
implementing the language core so that extension of the grammar becomes essentially trivial.

All told, we spent about 30 hours each writing code. Much of this time was spent thinking about Haskell and Scheme rather than actually typing. We were initially less ambitious about the scope of our implementation, but we found that many of the core features of Scheme are tightly integrated in such a way that one is not very interesting without the others. In some sense, our choice of which features to implement arose organically from the structure of the language.

This is a good place to note that none of our interpreters are particularly rich in features. None of them support multi-line expressions or loading of external programs, and the Haskell implementation doesn’t allow the user to delete text while working in the interactive interpreter. Also, the Haskell version supports only integers (no floats). In the interest of getting all three interpreters done by the deadline, we have left these details for another day.

We decided to implement the interpreters in roughly the same way across languages, proceeding in the following way:

1. **Interpreter Core** - Build the very basics of the interpreter based on wishful thinking. Spending time deciding what we want to come out of the parser would make our job easier later.

2. **Parser** - Now that we know what the input to the interpreter needs to look like, we have to produce it. We took different approaches to parser implementation in the three languages, but, in each case, the goal was the same.

3. **REPL** - Given just a parser and and interpreter, the idea of scope has no meaning. Scheme expressions may append to or update global state, and the REPL allows us to persist that state from evaluation to evaluation.

4. **Language Features** - Finally, once the interpreter is fully functional, we add more special forms and built-in functions.

**Architecture**

In order to keep things simple, we have decided (with the exception of the Parser section) to focus on the Haskell implementation here; however, we will periodically bring in details of the Ruby and Python implementations in order to demonstrate the differences.

**REPL**

The REPL, or Read Eval Print Loop, is the point of contact for the user of the interpreter. The REPL’s first parameter is an environment (symbol table), which stores
global bindings to be used during evaluation. The REPL reads input from stdin, runs it through the parser, and sends the result off to eval. The result of eval is printed, and the REPL calls itself recursively, prompting the user for more input. The key here is that changes to global state made inside a call to eval should persist between calls back into the REPL. The Haskell version can be seen in Figure 3.

Another thing to note is that I/O in Haskell is handled monadically, so all I/O must be done from inside a do block (or with explicit calls to >>=). The consequence of this is that run must also return an IO monad.

Parser

For the parser, our three implementations took distinctly different approaches, outlined as follows:

Python

This parser is built by hand, with help from Peter Norvig’s lis.py implementation. In comparison to the others, it is quite simple. It tokenizes the Scheme program into a list of strings, then it steps through the list. If the current token is (, the parser creates a new list and, until it sees ), calls itself recursively on each successive token, appending the result to that list. Broadly, the parser destroys the original list of tokens while simultaneously building a nested list to represent the AST. This approach is quite simple to implement and really highlights the convenience of Scheme’s homoiconicity.

Ruby

The Ruby parser was constructed using a parser generator called Treetop, which takes a parser specification in the form of a set of rules and generates some nasty looking Ruby code to do the actual work. This process is similar to the tradition yacc/bison approach to parsing, but with an object oriented twist. The parser that Treetop generates actually returns SyntaxNode objects, or instances of specialized classes which extend SyntaxNode.
These objects can then be processed into whatever form is required by the interpreter using a user-defined member function called \texttt{to\_lisp}. Treetop’s documentation is rather terse, but there is a tutorial online which parses a simplified subset of the Scheme grammar\footnote{4}. I used this in order to glean the general architecture. The disadvantages of this are a somewhat steep learning curve and that the errors one receives while debugging are somewhat opaque. However, the final parser is very easy to read and extend, and, once the correct emission functions are defined, it more or less stays out of the way of the interpreter.

Haskell

For our Parsec parser, we began with a skeleton derived from the parser provided by the fourth CMPS112 homework assignment\footnote{5}. This worked great for a while, until we noticed that the reserved words \texttt{let} and \texttt{lambda} were in conflict. After some deliberation, we decided that the parser was behaving deterministically (like a DFA). That is, upon reading an ‘l’, the parser would land in a state corresponding to the keyword \texttt{let}, falling off the machine on any character but an ‘e’. This was extremely confusing until we realized that we weren’t using Parsec to its full capabilities. Parsec contains a function called \texttt{makeTokenParser} which generates custom token parsers based on a user defined configuration as seen in Figure 4. Using the token parsers in this way gave us the lookahead/nondeterminism that we needed and was the major breakthrough that led to the stabilization of our parser. Ultimately, the parser emits instances of the algebraic datatype \texttt{SchVal}; individual expressions are represented by Haskell lists wrapped in the \texttt{List} constructor.

State

In order to store variables and custom function definitions, some idea of program state is needed. That is, each trip through the REPL should have access to global bindings defined by previous evaluations. Also, Scheme is lexically scoped, which means that bindings local to a block of code should only be available within that block. When the body of a function is evaluated, a local scope is created by merging the enclosing program state with the bindings between the function’s formal parameters and its arguments. This state is then passed back into \texttt{eval} with the body of the function.

Ruby and Python make this very simple. Each language comes with a heterogeneous and mutable dictionary/hash table which can store functions, values, lists, etc. without any extra work on the programmer’s part. For example, when a function is defined, we simply insert a native closure (lambda) into the dictionary, whose body is a call to \texttt{eval} and whose formal parameters are bound to those of the function in question. When a function call is evaluated, the name of the function is looked up in the dictionary and, if it exists, the corresponding closure is called with the \texttt{eval}ed arguments.

In Haskell, the corresponding data structure is the map. It is still an associative list in terms of functionality, but, unlike its dynamic cousins, it must be homogeneous and
def = emptyDef { P.reservedNames = ["quote", "if", "define", "let", "lambda", "#t", "#f"],
    P.opStart = oneOf "*/+-<><eq?=",
    P.opLetter = oneOf "*/+-<><eq?="
}

P.TokenParser { P.reserved = m_reserved,
    P.reservedOp = m_reservedOp,
    P.stringLiteral = m_stringLiteral,
    P.integer = m_integer,
    P.float = m_float,
    P.parens = parens,
    P.lexeme = lexeme } = P.makeTokenParser def

Figure 4: Custom token parser generation

it is not mutable. The first part is handled by our SchVal abstraction; the environment is simply a map from String to SchVal. This allows us to store values, functions, lists, etc. without issue. The lack of mutation is taken care of monadically. By carrying out evaluation from inside the State monad, we can effectively simulate a mutable program state using monadic operations.

Internally, the State monad contains a tuple comprising a value and a map. Stateful calculations are carried out by calls to a suite of related functions:

- **runState** - returns the whole tuple
- **evalState** - returns only the value (the first element of the tuple)
- **execState** - returns only the map (the second element of the tuple)

Each of these functions takes a State monad and an initial state (i.e. map) as arguments. For this reason, eval is designed to return a State so that its result can be passed as the first argument to runState. The second argument is simply our initial global environment. In our implementation, runState is called from run because both the value and the environment are needed (for printing and calling back into the REPL, respectively). We go into more detail on this below.

**Eval/apply**

Canonically, the core of the Scheme interpreter is a mutual recursion between two functions: eval and apply. Eval reads in an expression and has access to local state. With the exception of certain special forms (i.e. define, let, if, etc.) and the base case (atoms), this expression is usually a function call. In this case, the function and its arguments are passed down to apply, where the arguments themselves are evaled, bound to the
apply :: SchVal -> [SchVal] -> Store -> SchVal
apply (Func f) args = f args
apply (Lambda (List params) body) args symTable =
  evalState (eval body) newTable
  where newTable = (Map.union localTable symTable)
    localTable = Map.fromList (zip (map ripSymbol params) args)
    ripSymbol (Symbol x) = x

(a) Haskell version of apply

# `apply` (no need for a separate function b/c ruby is awesome)
else
  proc = list?(car(expr)) ? evalSch(car(expr), env) : env[car(expr)]
  args = cdr(expr).map { |a| evalSch(a, env) }
  # the call to eval is inside the closure
  return proc.call(*args) unless proc.nil?
  raise NameError, "function `#{car(expr)}` not defined"
end

(b) Catch-all “apply” case at the end of Ruby eval

formal parameters of the function, and the body is then passed into eval. This process
eventually bottoms out with a value, which is passed back up the stack and ultimately
printed. It should be clear that this architecture is perfectly suited to the s-expression
structure of Scheme code. If the s-expression is an atom its value is returned, if it is
an special form it is treated accordingly, and if it is a function call it is passed along to
apply.

In practice, this architecture can be modified for convenience without loss of ex-
pressive power. For example, neither of our dynamic implementations have a separate
apply function. This is primarily due to various syntactical conveniences which these
languages offer. The process of augmenting the current environment with the local
bindings amounts to the mere joining of two hash tables, and the Ruby/Python closures
corresponding to any function are easily applied to the argument list. The call back into
eval is actually hidden inside the body of the closure!

In Haskell, the process is similar, with one significant complication. Our environment
is kept by the State monad, and we need access to the symbol table itself in order to
augment it with bindings local to the function invocation. Also, eval must return a State
monad, while apply returns a SchVal. This means that we must evaluate the function’s
arguments from a do block inside eval and pop out of the State monad before passing
the eval-ed arguments and the symbol table itself to apply along with the function. In
apply, we construct a local symbol table by one-to-one correspondence between formal
parameters and arguments. Next, we augment the enclosing symbol table with the local
one, re-entering monad space with a new call to `evalState`, which in turn enters `eval`. The call to `evalState` is central to our derivation of lexical scope. We need to call one of the three monadic evaluation functions in order to perform the computation, but we only want the value back. The local state is thrown away after the computation is complete. This may seem complicated, but all the real complexity takes place “under the hood”; the final implementation is surprising clear. Figure 7 shows the Ruby and Haskell versions of “apply”.

Conclusions

Although we saw many advantages to the dynamic languages in terms of just getting something working, we found that Haskell’s type system and monad facilities encourage the use of well-defined abstractions in accomplishing any programming task. As a result, given some prior knowledge of Haskell, it is easier to understand why and how our Haskell implementation is correct. `Eval` contains a different pattern for every Scheme structure, and the path through the interpreter arises naturally from the type definitions and monad facilities used therein. Whereas Ruby and Python allowed us a certain degree of latitude in our handling of `eval/apply`, Haskell drove us toward an implementation that is more in line with the canonical implementation and more clearly displays its expressive power.

Learning how to interpret Scheme was an enlightening experience in its own right. Once we gained an understanding of Scheme’s s-expression structure, we realized that much of the complexity that we both associate with interpretation/compilation is essentially absent from Scheme. The bulk of the work in writing the Scheme interpreter lies in devising a way to represent these s-expressions in the implementation language. Scheme’s (and Lisp’s in general) ratio of expressive power to syntactic complexity seems to us unmatched in modern programming languages.

References


