Write Yourself a Scheme in 48 Hours
A Haskell Tutorial

Author: Jonathan Tang

Contents

1 Overview

Most Haskell tutorials on the web seem to take a language-reference-manual approach to teaching. They show you the syntax of the language, a few language constructs, and then have you construct a few simple functions at the interactive prompt. The "hard stuff" of how to write a functioning, useful program is left to the end, or sometimes omitted entirely.

This tutorial takes a different tack. You'll start off with command-line arguments and parsing, and progress to writing a fully-functional Scheme interpreter that implements a good-sized subset of R5RS Scheme. Along the way, you'll learn Haskell's I/O, mutable state, dynamic typing, error handling, and parsing features. By the time you finish, you should be fairly fluent in both Haskell and Scheme.

There're two main audiences targetted by this tutorial:

1. People who already know Lisp or Scheme and want to learn Haskell; and

2. People who don't know any programming language, but have a strong quantitative background and are familiar with computers.

The second group will likely find this challenging, as I gloss over several Scheme and general programming concepts to stay focused on the Haskell. A good textbook like Structure and Interpretation of Computer Programs or The Little Schemer may help a lot here.

Users of a procedural or object-oriented language like C, Java, or Python should beware, however: You'll have to forget most of what you already know about programming. Haskell is *completely* different from those languages, and requires a different way of thinking about programming. It's best to go into this tutorial with a blank slate and try not to compare Haskell to imperative languages, because many concepts in them (e.g. classes, functions, `return`) mean totally different things in Haskell.

Since each lesson builds on the code written for the previous one, it's probably best to go through the lessons in order.

This tutorial assumes that you'll be using GHC as your Haskell compiler. It may work with e.g. Hugs, but it hasn't been tested at all, and you may need to download additional libraries.

2 First steps

You'll want to download GHC from http://www.haskell.org/ghc/. A binary package is probably easiest, unless you really know what you're doing. It should download and install like any other software package. This tutorial was developed on Linux, but everything should also work on Windows as long as you know how to use the DOS command line.

There is a pretty good haskell-mode for Emacs, including syntax highlighting and automatic indentation. Windows users can use Notepad or any other text editor: Haskell syntax is fairly Notepad-friendly, though you have to be careful with the indentation.

Now, it's time for your first Haskell program. This program will read a name off the command line and then print a greeting. Create a file ending in ".hs" and type the following text:

```
module Main where
import System.Environment

main :: IO ()
main = do args <- getArgs
          putStrLn ("Hello, " ++ args !! 0)
```

Let's go through this code. The first two lines specify that we'll be creating a module named Main that imports the System module. Every Haskell program begins with an action called `main` in a module named `Main`. That module may import others, but it must be present for the compiler to generate an executable file. Haskell is case-sensitive: module names are always capitalized, declarations always uncapitalized.

The line `main :: IO ()` is a type declaration: it says that the action `main` has type `IO ()`. Type declarations in Haskell are optional: the compiler figures them out automatically, and only complains if they differ from what you've specified. In this tutorial, I specify the types of all declarations explicitly, for clarity. If you're following along at home, you may want to omit them, because it's less to change as we build our program.

The IO type is an instance of something called a *monad*, which is a scary name for a not-so-scary concept. Basically, a monad is a way of saying "there's some extra information attached to this value, which most functions don't need to worry about". In this example, the "extra information" is the fact that this action performs IO, and the basic value is nothing, represented as `()`. Monadic values are often called "actions", because the easiest way to think about the IO monad is a sequencing of actions that each might affect the outside world.

Haskell is a declarative language: instead of giving the computer a sequence of instructions to carry out, you give it a collection of definitions that tell it how to perform every function it might need. These definitions use various compositions of actions and functions. The compiler figures out an execution path that puts everything together.

To write one of these definitions, you set it up as an equation. The left hand side defines a name, and optionally one or more *patterns* (explained later) that will bind variables. The right hand side defines some composition of other definitions that tells the computer what to do when it encounters the name. These equations behave just like ordinary equations in algebra: you can always substitute the right hand side for the left within the text of the program, and it'll evaluate to the same value. Called "referential transparency", this property makes it significantly easier to reason about Haskell programs than other languages.

How will we define our `main` action? We know that it must be an `IO ()` action, and that we want it to read the command line args and print some output. There are two ways to create an IO action:

1. Lift an ordinary value into the IO monad, using the return function; or

2. Combine two existing IO actions.

Since we want to do two things, we'll take the second approach. The built-in action getArgs reads the command-line arguments and stores them in a list of strings. The built-in function putStrLn takes a string and writes it to the console.

To combine them, we use a do-block. A do-block consists of a series of lines, all lined up with the the first non-whitespace character after the do. Each line can have one of two forms:

1. `name <- action`; or

2. `action`.

The first form takes the result of `action` and binds it to `name`. For example, if the type of the action is `IO [String]` (an IO action returning a list of strings, as with `getArgs`), then `name` will be bound to the list of strings returned. The second form just executes the action, sequencing it with the previous line through the >> (pronounced "bind") operator. This operator has different semantics for each monad: in the case of the IO monad, it executes the actions sequentially, performing whatever external side-effects that result. Because the semantics of this composition depend upon the particular monad used, you cannot mix actions of different monad types in the same do-block.

Of course, these actions may themselves be the result of functions or complicated expressions. In this example, we first take index 0 of the argument list (`args !! 0`), concatenate it onto the end of the string (`"Hello, " ++`), and finally pass that to `putStrLn` for IO sequencing. Strings are lists of characters in Haskell, so you can use any of the list functions and operators on them. A full table of the standard operators and their precedences follows:

Operator(s)	**Prec.**	**Assoc.**	**Description**
.	9	Right	Function composition

Operator(s)	Prec.	Assoc.	Description
`!!`	9	Left	List indexing
`^`, `^^`, `**`	8	Right	Exponentiation (integer, fractional, and floating-point)
`*`, `/`	7	Left	Multiplication, Division
`+`, `-`	6	Left	Addition, Subtraction
`:`	5	Right	Cons (list construction)
`+`	5	Right	List Concatenation
‘elem‘, ‘notElem‘	4	Left	List Membership
`==`, `/=`, `<`, `<=`, `>=`, `>`	4	Left	Equals, Not-equals, and other relation operators
`&&`	3	Right	Logical And
`||`	2	Right	Logical Or
`>>`, `>>=`	1	Left	Monadic Bind, Monadic Bind (piping value to next function)
`=<<`	1	Right	Reverse Monadic Bind (same as above, but arguments reversed)
`$`	0	Right	Infix Function Application (same as `f x`, but right-associative instead of left)

To compile and run the program, try something like this:

```
$ ghc -o hello_you listing2.hs
$ ./hello_you Jonathan
Hello, Jonathan
```

The `-o` option specifies the name of the executable you want to create, and then you just specify the name of the Haskell source file.

2.1 Exercises

1. Change the program so it reads *two* arguments from the command line, and prints out a message using both of them.

2. Change the program so it performs a simple arithmetic operation on the two arguments and prints out the result. You can use read to convert a string to a number, and show to convert a number back into a string. Play around with different operations.

3. `getLine` is an IO action that reads a line from the console and returns it as a string. Change the program so it prompts for a name, reads the name, and then prints that instead of the command line value

3 Parsing

3.1 Writing a Simple Parser

Now, let's try writing a very simple parser. We'll be using the Parsec library, which comes with GHC but may need to be downloaded separately if you're using another compiler.

Start by adding this line to the import section:

```
import Text.ParserCombinators.Parsec hiding (spaces)
```

This makes the Parsec library functions available to us, except the `spaces` function, whose name conflicts with a function that we'll be defining later.

Now, we'll define a parser that recognizes one of the symbols allowed in Scheme identifiers:

```
symbol :: Parser Char
symbol = oneOf "!$%&|*+-/:<=>?@^_~"
```

This is another example of a monad: in this case, the "extra information" that is being hidden is all the info about position in the input stream, backtracking record, first and follow sets, etc. Parsec takes care of all of that for us. We need only use the Parsec library function oneOf, and it'll recognize a single one of any of the characters in the string passed to it. Parsec provides a number of pre-built parsers: for example, letter and digit are library functions. And as you're about to see, you can compose primitive parsers into more sophisticated productions.

Let's define a function to call our parser and handle any possible errors:

```
readExpr :: String -> String
readExpr input = case parse symbol "lisp" input of
    Left err -> "No match: " ++ show err
    Right val -> "Found value"
```

As you can see from the type signature, readExpr is a function (`->`) from a String to a String. We name the parameter `input`, and pass it, along with the `symbol` action we defined above and the name of the parser (`lisp`), to the Parsec function parse.

Parse can return either the parsed value or an error, so we need to handle the error case. Following typical Haskell convention, Parsec returns an Either data type, using the Left constructor to indicate an error and the Right one for a normal value.

We use a `case ... of` construction to match the result of `parse` against these alternatives. If we get a Left value (an error), then we bind the error itself to `err` and return `"No match"` with the string representation of the error. If we get a Right value, we bind it to `val`, ignore it, and return the string `"Found value"`.

The `case ... of` construction is an example of pattern matching, which we will see in much greater detail later on.

Finally, we need to change our main function to call readExpr and print out the result:

```
main :: IO ()
main = do args <- getArgs
          putStrLn (readExpr (args !! 0))       -- changed
```

To compile and run this, you need to specify `-package parsec` on the command line, or else there will be link errors. For example:

```
$ ghc -package parsec -o simple_parser listing3.1.hs
$ ./simple_parser $
Found value
$ ./simple_parser a
No match: "lisp" (line 1, column 1):
unexpected "a"
```

3.2 Whitespace

Next, we'll add a series of improvements to our parser that'll let it recognize progressively more complicated expressions. The current parser chokes if there's whitespace preceding our symbol:

```
$ ./simple_parser "   %"
No match: "lisp" (line 1, column 1):
unexpected " "
```

Let's fix that, so that we ignore whitespace.

First, lets define a parser that recognizes any number of whitespace characters. Incidentally, this is why we included the `hiding (spaces)` clause when we imported Parsec: there's already a function spaces in that library, but it doesn't quite do what we want it to. (For that matter, there's also a parser called lexeme that does exactly what we want, but we'll ignore that for pedagogical purposes.)

```
spaces :: Parser ()
spaces = skipMany1 space
```

Just as functions can be passed to functions, so can actions. Here we pass the Parser action space to the Parser action skipMany1, to get a Parser that will recognize one or more spaces.

Now, let's edit our parse function so that it uses this new parser:

```
readExpr input = case parse (spaces >> symbol) "lisp" input of  -- changed
    Left err -> "No match: " ++ show err
    Right val -> "Found value"
```

We touched briefly on the >> ("bind") operator in First Steps, where we mentioned that it was used behind the scenes to combine the lines of a do-block. Here, we use it explicitly to combine our whitespace and symbol parsers. However, bind has completely different semantics in the Parser and IO monads. In the Parser monad, bind means "Attempt to match the first parser, then attempt to match the second with the remaining input, and fail if either fails." In general, bind will have wildly different effects in different monads; it's intended as a general way to structure computations, and so needs to be general enough to accomodate all the different types of computations. Read the documentation for the monad to figure out precisely what it does.

Compile and run this code. Note that since we defined spaces in terms of `skipMany1`, it will no longer recognize a plain old single character. Instead you *have to* preceed a symbol with some whitespace. We'll see how this is useful shortly:

```
$ ghc -package parsec -o simple_parser listing3.2.hs
$ ./simple_parser "   %" Found value
$ ./simple_parser %
No match: "lisp" (line 1, column 1):
unexpected "%"
expecting space
$ ./simple_parser "   abc"
No match: "lisp" (line 1, column 4):
unexpected "a"
expecting space
```

3.3 Return Values

Right now, the parser doesn't *do* much of anything - it just tells us whether a given string can be recognized or not. Generally, we want something more out of our parsers: we want them to convert the input into a data structure that we can traverse easily. In this section, we learn how to define a data type, and how to modify our parser so that it returns this data type.

First, we need to define a data type that can hold any Lisp value:

```
data LispVal = Atom String
             | List [LispVal]
             | DottedList [LispVal] LispVal
             | Number Integer
             | String String
             | Bool Bool
```

This is an example of an *algebraic data type*: it defines a set of possible values that a variable of type LispVal can hold. Each alternative (called a *constructor* and separated by `|`) contains a tag for the constructor along with the type of data that that constructor can hold. In this example, a LispVal can be:

1. An `Atom`, which stores a String naming the atom;
2. A `List`, which stores a list of other LispVals (Haskell lists are denoted by brackets);
3. A `DottedList`, representing the Scheme form `(a b . c)`. This stores a list of all elements but the last, and then stores the last element as another field;
4. A `Number`, containing a Haskell Integer;
5. A `String`, containing a Haskell String; or
6. A `Bool`, containing a Haskell boolean value.

Constructors and types have different namespaces, so you can have both a constructor named String and a type named String. Both types and constructor tags always begin with capital letters.

Next, let's add a few more parsing functions to create values of these types. A string is a double quote mark, followed by any number of non-quote characters, followed by a closing quote mark:

```
parseString :: Parser LispVal
parseString = do char '"'
                 x <- many (noneOf "\"")
                 char '"'
                 return $ String x
```

We're back to using the do-notation instead of the `>>` operator. This is because we'll be retrieving the value of our parse (returned by `many (noneOf "\"")`) and manipulating it, interleaving some other parse operations in the meantime. In general, use `>>` if the actions don't return a value, `>>=` if you'll be immediately passing that value into the next action, and do-notation otherwise.

Once we've finished the parse and have the Haskell String returned from `many`, we apply the String constructor (from our LispVal data type) to turn it into a LispVal. Every constructor in an algebraic data type also acts like a function that turns its arguments into a value of its type. It also serves as a pattern that can be used in the left-hand side of a pattern-matching expression; we saw an example of this in Writing a Simple Parser when we matched our parser result against the two constructors in the Either data type.

We then apply the built-in function return to lift our LispVal into the Parser monad. Remember, each line of a do-block must have the same type, but the result of our String constructor is just a plain old LispVal. Return lets us wrap that up in a Parser action that consumes no input but returns it as the inner value. Thus, the whole parseString action will have type Parser LispVal.

The `$` operator is infix function application: it's the same as if we'd written `return (String x)`, but `$` is right-associative, letting us eliminate some parentheses.

Now let's move on to Scheme variables. An atom is a letter or symbol, followed by any number of letters, digits, or symbols:

```
parseAtom :: Parser LispVal
parseAtom = do first <- letter <|> symbol
               rest <- many (letter <|> digit <|> symbol)
               let atom = [first] ++ rest
               return $ case atom of
                          "#t" -> Bool True
                          "#f" -> Bool False
                          otherwise -> Atom atom
```

Here, we introduce another Parsec combinator, the choice operator <|>. This tries the first parser, then if it fails, tries the second. If either succeeds, then it returns the value returned by that parser. The first parser must fail before it consumes any input: we'll see later how to implement backtracking.

Once we've read the first character and the rest of the atom, we need to put them together. The "let" statement defines a new variable "atom". We use the list concatenation operator `++` for this. Recall that `first` is just a single character, so we convert it into a singleton list by putting brackets around it. If we'd wanted to create a list containing many elements, we need only separate them by commas.

Then we use a case statement to determine which LispVal to create and return, matching against the literal strings for true and false. The `otherwise` alternative is a readability trick: it binds a variable named `otherwise`, whose value we ignore, and then always returns the value of `atom`.

Finally, we create one more parser, for numbers. This shows one more way of dealing with monadic values:

```
parseNumber :: Parser LispVal
parseNumber = liftM (Number . read) $ many1 digit
```

It's easiest to read this backwards, since both function application (`$`) and function composition (.) associate to the right. The parsec combinator many1 matches one or more of its argument, so here we're matching one or more digits. We'd like to construct a number LispVal from the resulting string, but we have a few type mismatches. First, we use the built-in function read to convert that string into a number. Then we pass the result to Number to get a LispVal. The function composition operator `.` creates a function that applies its right argument and then passes the result to the left argument, so we use that to combine the two function applications.

Unfortunately, the result of `many1 digit` is actually a `Parser String`, so our combined `Number . read` still can't operate on it. We need a way to tell it to just operate on the value inside the monad, giving us back a Parser LispVal. The standard function `liftM` does exactly that, so we apply liftM to our `Number . read` function, and then apply the result of that to our Parser.

We also have to import the Monad module up at the top of our program to get access to liftM:

```
import Monad
```

This style of programming - relying heavily on function composition, function application, and passing functions to functions - is very common in Haskell code. It often lets you express very complicated algorithms in a single line, breaking down intermediate steps into other functions that can be combined in various ways. Unfortunately, it means that you often have to read Haskell code from right-to-left and keep careful track of the types. We'll be seeing many more examples throughout the rest of the tutorial, so hopefully you'll get pretty comfortable with it.

Let's create a parser that accepts either a string, a number, or an atom:

```
parseExpr :: Parser LispVal
parseExpr = parseAtom
        <|> parseString
        <|> parseNumber
```

And edit readExpr so it calls our new parser:

```
readExpr :: String -> String
readExpr input = case parse parseExpr "lisp" input of   -- changed
    Left err -> "No match: " ++ show err
    Right _ -> "Found value"
```

Compile and run this code, and you'll notice that it accepts any number, string, or symbol, but not other strings:

```
$ ghc -package parsec -o simple_parser listing3.3.hs
$ ./simple_parser '"this is a string"'
```

```
Found value
$ ./simple_parser 25 Found value
$ ./simple_parser symbol
Found value
$ ./simple_parser '(symbol)'
No match: "lisp" (line 1, column 1):
unexpected "("
expecting letter, "\"" or digit
```

3.4 Exercises

1. Rewrite parseNumber using

   A. do-notation; and

   B. explicit sequencing with the >>= operator.

2. Our strings aren't quite R5RS compliant, because they don't support escaping of internal quotes within the string. Change parseString so that `\"` gives a literal quote character instead of terminating the string. You may want to replace `noneOf "\""` with a new parser action that accepts *either* a non-quote character *or* a backslash followed by a quote mark.

3. Modify the previous exercise to support `\n`, `\r`, `\t`, `\\`, and any other desired escape characters.

4. Change parseNumber to support the Scheme standard for different bases. You may find the readOct and readHex functions useful.

5. Add a Character constructor to LispVal, and create a parser for character literals as described in R5RS.

6. Add a Float constructor to LispVal, and support R5RS syntax for decimals. The Haskell function readFloat may be useful.

7. Add data types and parsers to support the full numeric tower of Scheme numeric types. Haskell has built-in types to represent many of these; check the Prelude. For the others, you can define compound types that represent eg. a Rational as a numerator and denominator, or a Complex as a real and imaginary part (each itself a Real number).

3.5 Recursive Parsers: Adding lists, dotted lists, and quoted datums

Next, we add a few more parser actions to our interpreter. Start with the parenthesized lists that make Lisp famous:

```
parseList :: Parser LispVal
parseList = liftM List $ sepBy parseExpr spaces
```

This works analogously to parseNumber, first parsing a series of expressions separated by whitespace (`sepBy parseExpr spaces`) and then apply the List constructor to it within the Parser monad. Note too that we can pass parseExpr to sepBy, even though it's an action we wrote ourselves.

The dotted-list parser is somewhat more complex, but still uses only concepts that we're already familiar with:

```
parseDottedList :: Parser LispVal
parseDottedList = do
    head <- endBy parseExpr spaces
    tail <- char '.' >> spaces >> parseExpr
    return $ DottedList head tail
```

Note how we can sequence together a series of Parser actions with `>>` and then use the whole sequence on the right hand side of a do-statement. The expression `char '.' >> spaces` returns a `Parser ()`, then combining that with parseExpr gives a Parser LispVal, exactly the type we need for the do-block.

Next, let's add support for the single-quote syntactic sugar of Scheme:

```
parseQuoted :: Parser LispVal
parseQuoted = do
    char '\''
    x <- parseExpr
    return $ List [Atom "quote", x]
```

Most of this is fairly familiar stuff: it reads a single quote character, reads an expression and binds it to x, and then returns `(quote x)`, to use Scheme notation. The Atom constructor works like an ordinary function: you pass it the String you're encapsulating, and it gives you back a LispVal. You can do anything with this LispVal that you normally could, like put it in a list.

Finally, edit our definition of parseExpr to include our new parsers:

```
parseExpr :: Parser LispVal
parseExpr = parseAtom
        <|> parseString
        <|> parseNumber
        <|> parseQuoted                                 -- changed
        <|> do char '('                                 -- changed
               x <- (try parseList) <|> parseDottedList -- changed
               char ')'                                 -- changed
               return x                                 -- changed
```

This illustrates one last feature of Parsec: backtracking. parseList and parseDottedList recognize identical strings up to the dot; this breaks the requirement that a choice alternative may not consume any input before failing. The try combinator attempts to run the specified parser, but if it fails, it backs up to the previous state. This lets you use it in a choice alternative without interfering with the other alternative.

Compile and run this code:

```
$ ghc -package parsec -o simple_parser listing3.4.hs
$ ./simple_parser "(a test)"
Found value
$ ./simple_parser "(a (nested) test)" Found value
$ ./simple_parser "(a (dotted . list) test)"
Found value
$ ./simple_parser "(a '(quoted (dotted . list)) test)"
Found value
$ ./simple_parser "(a '(imbalanced parens)"
No match: "lisp" (line 1, column 24):
unexpected end of input
expecting space or ")"
```

Note that by referring to parseExpr within our parsers, we can nest them arbitrarily deep. Thus, we get a full Lisp reader with only a few definitions. That's the power of recursion.

3.6 Exercises

1. Add support for the backquote syntactic sugar: the Scheme standard details what it should expand into (quasiquote/unquote).

2. Add support for vectors. The Haskell representation is up to you: GHC does have an Array data type, but it can be difficult to use. Strictly speaking, a vector should have constant-time indexing and updating, but destructive update in a purely functional language is difficult. You may have a better idea how to do this after the section on set!, later in this tutorial.

3. Instead of using the try combinator, left-factor the grammar so that the common subsequence is its own parser. You should end up with a parser that matches a string of expressions, and one that matches either nothing or a dot and a single expressions. Combining the return values of these into either a List or a DottedList is left as a (somewhat tricky) exercise for the reader: you may want to break it out into another helper function

4 Evaluation, Part 1

4.1 Beginning the Evaluator

Currently, we've just been printing out whether or not we recognize the given program fragment. We're about to take the first steps towards a working Scheme interpreter: assigning values to program fragments. We'll be starting with baby steps, but fairly soon you'll be progressing to doing working computations.

Let's start by telling Haskell how to print out a string representation of the various possible LispVals:

```
showVal :: LispVal -> String
showVal (String contents) = "\"" ++ contents ++ "\""
showVal (Atom name) = name
showVal (Number contents) = show contents
showVal (Bool True) = "#t"
showVal (Bool False) = "#f"
```

This is our first real introduction to pattern matching. Pattern matching is a way of destructuring an algebraic data type, selecting a code clause based on its constructor and then binding the components to variables. Any constructor can appear in a pattern; that pattern matches a value if the tag is the same as the value's tag and all subpatterns match their corresponding components. Patterns can be nested arbitrarily deep, with matching proceeding in an inside-to-outside, left-to-right order. The clauses of a function definition are tried in textual order, until a pattern matches. If this is confusing, you'll see some examples of deeply-nested patterns when we get further into the evaluator.

For now, you only need to know that each clause of the above definition matches one of the constructors of LispVal, and the right-hand side tells what to do for a value of that constructor.

The List and DottedList clauses work similarly, but we need to define a helper function `unwordsList` to convert the contained list into a string:

```
showVal (List contents) = "(" ++ unwordsList contents ++ ")"
showVal (DottedList head tail) = "(" ++ unwordsList head
                                    ++ " . " ++ showVal tail
                                    ++ ")"
```

The unwordsList function works like the Haskell Prelude's unwords function, which glues together a list of words with spaces. Since we're dealing with a list of LispVals instead of words, we define a function that first converts the LispVals into their string representations and then applies unwords to it:

```
unwordsList :: [LispVal] -> String
unwordsList = unwords . map showVal
```

Our definition of unwordsList doesn't include any arguments. This is an example of *point-free style*: writing definitions purely in terms of function composition and partial application, without regard to individual values or “points”. Instead, we define it as the composition of a couple built-in functions. First, we partially-apply map to showVal, which creates a function that takes a list of LispVals and returns a list of their string representations. Haskell functions are *curried*: this means that a function of two arguments, like `map`, is really a function that returns a function of one argument. As a result, if you supply only a single argument, you get back a function one argument that you can pass around, compose, and apply later. In this case, we compose it with unwords: `map showVal` converts a list of LispVals to a list of their string representations, and then `unwords` joins the result together with spaces.

We used the function show above. This standard Haskell function lets you convert any type that’s an instance of the class Show into a string. We’d like to be able to do the same with LispVal, so we make it into a member of the class Show, defining its “show” method as showVal:

```
instance Show LispVal where show = showVal
```

A full treatment of typeclasses is beyond the scope of this tutorial; you can find more information in other tutorials and the Haskell 98 report.

Let’s try things out by changing our readExpr function so it returns the string representation of the value actually parsed, instead of just `"Found value"`:

```
readExpr input = case parse parseExpr "lisp" input of
    Left err -> "No match: " ++ show err
    Right val -> "Found " ++ show val   -- changed
```

And compile and run...

```
$ ghc -package parsec -o parser listing4.1.hs
$ ./parser "(1 2 2)"
Found (1 2 2)
$ ./parser "'(1 3 (\"this\" \"one\"))"
Found (quote (1 3 ("this" "one")))
```

4.2 Beginnings of an evaluator: Primitives

Now, we start with the beginnings of an evaluator. The purpose of an evaluator is to map some “code” data type into some “data” data type, the result of the evaluation. In Lisp, the data *types* for both code and data are the same, so our evaluator will return a LispVal. Other languages often have more complicated code structures, with a variety of syntactic forms.

Evaluating numbers, strings, booleans, and quoted lists is fairly simple: return the datum itself.

```
eval :: LispVal -> LispVal
eval val@(String _) = val
eval val@(Number _) = val
eval val@(Bool _) = val
eval (List [Atom "quote", val]) = val
```

This introduces a new type of pattern. The notation `val@(String _)` matches against any LispVal that’s a string and then binds `val` to the *whole* LispVal, and not just the contents of the String constructor. The result has type LispVal instead of type String. The underbar is the “don’t care” variable, matching any value yet not binding it to a variable. It can be used in any pattern, but is most useful with `@`-patterns (where you bind the variable to the whole pattern) and with simple constructor-tests where you’re just interested in the tag of the constructor.

The last clause is our first introduction to nested patterns. The type of data contained by `List` is `[LispVal]`, a list of LispVals. We match *that* against the specific two-element list `[Atom "quote", val]`, a list where the first element is the symbol “quote” and the second element can be anything. Then we return that second element.

Let’s integrate `eval` into our existing code. Start by changing readExpr back so it returns the expression instead of a string representation of the expression:

```
readExpr :: String -> LispVal                                -- changed
readExpr input = case parse parseExpr "lisp" input of
    Left err -> String $ "No match: " ++ show err       -- changed
    Right val -> val                                     -- changed
```

And then change our main function to read an expression, evaluate it, convert it to a string, and print it out. Now that we know about the `>>=` monad sequencing operator and the function composition operator, let's use them to make this a bit more concise:

```
main :: IO ()
main = getArgs >>= putStrLn . show . eval . readExpr . (!! 0)
```

Here, we take the result of the getArgs action (a list of strings) and pass it into the composition of:

1. Take the first value (`(!! 0)`). This notation is known as an *operator section*: it's telling the compiler to partially-apply the list indexing operator to 0, giving you back a function that takes the first element of whatever list it's passed.
2. Parse it (`readExpr`);
3. Evaluate it (`eval`);
4. Convert it to a string (`show`); and finally
5. Print it (`putStrLn`).

Compile and run the code the normal way:

```
$ ghc -package parsec -o eval listing4.2.hs
$ ./eval "'atom"
atom
$ ./eval 2
2
$ ./eval '"a string"'
"a string"
$ ./eval "(+ 2 2)"

Fail: listing6.hs:83: Non-exhaustive patterns in function eval
```

We still can't do all that much useful with the program (witness the failed (+ 2 2) call), but the basic skeleton is in place. Soon, we'll be extending it with some functions to make it useful.

4.3 Adding basic primitives

Next, we'll improve our Scheme so we can use it as a simple calculator. It's still not yet a "programming language", but it's getting close.

Begin by adding a clause to eval to handle function application. Remember that all clauses of a function definition must be placed together and are evaluated in textual order, so this should go after the other eval clauses:

```
eval (List (Atom func : args)) = apply func $ map eval args
```

This is another nested pattern, but this time we match against the cons operator `:` instead of a literal list. Lists in Haskell are really syntactic sugar for a chain of cons applications and the empty list: `[1, 2, 3, 4] = 1:(2:(3:(4:[])))`. By pattern-matching against cons itself instead of a literal list, we're saying "give me the rest of the list" instead of "give me the second element of the list". For example, if we passed (+ 2 2) to eval, func would be bound to + and args would be bound to [Number 2, Number 2].

The rest of the clause consists of a couple functions we've seen before and one we haven't defined yet. We have to recursively evaluate each argument, so we map eval over the args. This is what lets us write compound expressions like (+ 2 (- 3 1) (* 5 4)). Then we take the resulting list of evaluated arguments, and pass it and the original function to apply:

```
apply :: String -> [LispVal] -> LispVal
apply func args = maybe (Bool False) ($ args) $ lookup func primitives
```

The built-in function lookup looks up a key (its first argument) in a list of pairs. However, lookup will fail if no pair in the list contains the matching key. To express this, it returns an instance of the built-in type Maybe. We use the function maybe to specify what to do in case of either success or failure. If the function isn't found, we return a Bool False value, equivalent to #f (we'll add more robust error-checking later). If it *is* found, we apply it to the arguments using ($ args), an operator section of the function application operator.

Next, we define the list of primitives that we support:

```
primitives :: [(String, [LispVal] -> LispVal)]
primitives = [("+", numericBinop (+)),
              ("-", numericBinop (-)),
              ("*", numericBinop (*)),
              ("/", numericBinop div),
              ("mod", numericBinop mod),
              ("quotient", numericBinop quot),
              ("remainder", numericBinop rem)]
```

Look at the type of primitives. It is a list of pairs, just like lookup expects, but the values of the pairs are *functions* (from [LispVal] to LispVal). In Haskell, you can easily store functions in other data structures, though the functions must all have the same type.

Also, the functions that we store are themselves the result of a function, numericBinop, which we haven't defined yet. This takes a primitive Haskell function (often an operator section) and wraps it with code to unpack an argument list, apply the function to it, and wrap the result up in our Number constructor.

```
numericBinop :: (Integer -> Integer -> Integer) -> [LispVal] -> LispVal
numericBinop op params = Number $ foldl1 op $ map unpackNum params

unpackNum :: LispVal -> Integer
unpackNum (Number n) = n
unpackNum (String n) = let parsed = reads n in
                          if null parsed
                            then 0
                            else fst $ parsed !! 0
unpackNum (List [n]) = unpackNum n
unpackNum _ = 0
```

As with R5RS Scheme, we don't limit ourselves to only two arguments. Our numeric operations can work on a list of any length, so (+ 2 3 4) = 2 + 3 + 4, and (- 15 5 3 2) = 15 - 5 - 3 - 2. We use the built-in function foldl1 to do this. It essentially changes every cons operator in the list to the binary function we supply, op.

Unlike R5RS Scheme, we're implementing a form of *weak typing*. That means that if a value can be interpreted as a number (like the string "2"), we'll use it as one, even if it's tagged as a string. We do this by adding a couple extra clauses to unpackNum. If we're unpacking a string, attempt to parse it with Haskell's built-in reads function, which returns a list of pairs of (parsed value, original value).

For lists, we pattern-match against the one-element list and try to unpack that. Anything else falls through to the next case.

If we can't parse the number, for any reason, we'll return 0 for now. We'll fix this shortly so that it signals an error.

Compile and run this the normal way. Note how we get nested expressions "for free" because we call eval on each of the arguments of a function:

```
$ ghc -package parsec -o eval listing7.hs
$ ./eval "(+ 2 2)" 4
$ ./eval "(+ 2 (-4 1))"
2
$ ./eval "(+ 2 (- 4 1))"
5
$ ./eval "(- (+ 4 6 3) 3 5 2)"
3
```

4.4 Exercises

1. Add primitives to perform the various type-testing functions of R5RS: symbol?, string?, number?, etc.

2. Change unpackNum so that it always returns 0 if the value is not a number, even if it's a string or list that could be parsed as a number.

3. Add the symbol-handling functions from R5RS. A symbol is what we've been calling an Atom in our data constructors

5 Intermezzo: Error Checking

Currently, there are a variety of places within the code where we either ignore errors or silently assign "default" values like `#f` or `0` that make no sense. Some languages - like Perl and PHP - get along fine with this approach. However, it often means that errors pass silently throughout the program until they become big problems, which means rather inconvenient debugging sessions for the programmer. We'd like to signal errors as soon as they happen and immediately break out of execution.

First, we need to import Control.Monad.Error to get access to Haskell's built-in error functions:

```
import Control.Monad.Error
```

Then, we should define a data type to represent an error:

```
data LispError = NumArgs Integer [LispVal]
               | TypeMismatch String LispVal
               | Parser ParseError
               | BadSpecialForm String LispVal
               | NotFunction String String
               | UnboundVar String String
               | Default String
```

This is a few more constructors than we need at the moment, but we might as well forsee all the other things that can go wrong in the interpreter later. Next, we define how to print out the various types of errors and make LispError an instance of Show:

```
showError :: LispError -> String
showError (UnboundVar message varname)  = message ++ ": " ++ varname
showError (BadSpecialForm message form) = message ++ ": " ++ show form
showError (NotFunction message func)    = message ++ ": " ++ show func
showError (NumArgs expected found)      =
    "Expected " ++ show expected ++ " args; found values " ++ unwordsList found
showError (TypeMismatch expected found) =
    "Invalid type: expected " ++ expected ++ ", found " ++ show found
showError (Parser parseErr)             = "Parse error at " ++ show parseErr

instance Show LispError where show = showError
```

Our next step is to make our error type into an instance of Error. This is necessary for it to work with GHC's built-in error handling functions. Being an instance of error just means that it must provide functions to create an instance either from a previous error message or by itself:

```
instance Error LispError where
     noMsg = Default "An error has occurred"
     strMsg = Default
```

Then we define a type to represent functions that may throw a LispError or return a value. Remember how parse used an Either data type to represent exceptions? We take the same approach here:

```
type ThrowsError = Either LispError
```

Type constructors are curried just like functions, and can also be partially applied. A full type would be `Either LispError Integer` or `Either LispError LispVal`, but we want to say `ThrowsError LispVal` and so on. We only partially apply Either to LispError, creating a type constructor ThrowsError that we can use on any data type.

`Either` is yet another instance of a monad. In this case, the "extra information" being passed between Either actions is whether or not an error occurred. Bind applies its function if the Either action holds a normal value, or passes an error straight through without computation. This is how exceptions work in other languages, but because Haskell is lazily-evaluated, there's no need for a separate control-flow construct. If bind determines that a value is already an error, the function is never called.

The Either monad also provides two other functions besides the standard monadic ones:

1. throwError, which takes an Error value and lifts it into the Left (error) constructor of an Either; and

2. catchError, which takes an Either action and a function that turns an error into another Either action. If the action represents an error, it applies the function, which you can use to e.g. turn the error value into a normal one via `return` or re-throw as a different error.

In our program, we'll be converting all of our errors to their string representations and returning that as a normal value. Let's create a helper function to do that for us:

```
trapError action = catchError action (return . show)
```

The result of calling trapError is another Either action which will always have valid (Right) data. We still need to extract that data from the Either monad so it can passed around to other functions:

```
extractValue :: ThrowsError a -> a
extractValue (Right val) = val
```

We purposely leave extractValue undefined for a Left constructor, because that represents a programmer error. We intend to use extractValue only after a catchError, so it's better to fail fast than to inject bad values into the rest of the program.

Now that we have all the basic infrastructure, it's time to start using our error-handling functions. Remember how our parser had previously just return a string saying `"No match"` on an error? Let's change it so that it wraps and throws the original ParseError:

```
readExpr :: String -> ThrowsError LispVal      -- changed
readExpr input = case parse parseExpr "lisp" input of
    Left err -> throwError $ Parser err        -- changed
    Right val -> return val                    -- changed
```

Here, we first wrap the original ParseError with the LispError constructor Parser, and then use the built-in function throwError to return that in our ThrowsError monad. Since readExpr now returns a monadic value, we also need to wrap the other case in a return function.

Next, we change the type signature of eval to return a monadic value, adjust the return values accordingly, and add a clause to throw an error if we encounter a pattern that we don't recognize:

```
eval :: LispVal -> ThrowsError LispVal
eval val@(String _) = return val
eval val@(Number _) = return val
eval val@(Bool _) = return val
eval (List [Atom "quote", val]) = return val
eval (List (Atom func : args)) = mapM eval args >>= apply func
eval badForm = throwError $ BadSpecialForm "Unrecognized special form" badForm
```

Since the function application clause calls eval (which now returns a monadic value) recursively, we need to change that clause. First, we had to change map to mapM, which maps a monadic function over a list of values, sequences the resulting actions together with bind, and then returns a list of the inner results. Inside the Error monad, this sequencing performs all computations sequentially but throws an error value if any one of them fails - giving you `Right [results]` on success, or `Left error` on failure. Then, we used the monadic "bind" operation to pass the result into the partially applied `apply func`, again returning an error if either operation failed.

Next, we change apply itself so that it throws an error if it doesn't recognize the function:

```
apply :: String -> [LispVal] -> ThrowsError LispVal
apply func args =
    maybe (throwError $ NotFunction "Unrecognized primitive function args" func)
          ($ args)
          (lookup func primitives)
```

We *didn't* add a return statement to the function application (`$ args`). We're about to change the type of our primitives, so that the function returned from the lookup itself returns a ThrowsError action:

```
primitives :: [(String, [LispVal] -> ThrowsError LispVal)]
```

And, of course, we need to change the numericBinop function that implements these primitives so it throws an error if there's only one argument:

```
numericBinop :: (Integer -> Integer -> Integer) -> [LispVal]
             -> ThrowsError LispVal
numericBinop op singleVal@[_] = throwError $ NumArgs 2 singleVal
numericBinop op params = mapM unpackNum params >>= return . Number . foldl1 op
```

We use an at-pattern to capture the single-value case because we want to include the actual value passed in for error-reporting purposes. Here, we're looking for a list of exactly one element, and we don't care what that element is. We also need to use mapM to sequence the results of unpackNum, because each individual call to unpackNum may fail with a TypeMismatch:

```
unpackNum :: LispVal -> ThrowsError Integer
unpackNum (Number n) = return n
unpackNum (String n) = let parsed = reads n in
```

```
                                if null parsed
                                  then throwError $ TypeMismatch "number" $ String n
                                  else return $ fst $ parsed !! 0
unpackNum (List [n]) = unpackNum n
unpackNum notNum = throwError $ TypeMismatch "number" notNum
```

Finally, we need to change our main function to use this whole big error monad. This can get a little complicated, because now we're dealing with *two* monads (Error and IO). As a result, we go back to do-notation, because it's nearly impossible to use point-free style when the result of one monad is nested inside another:

```
main :: IO ()
main = do
    args <- getArgs
    evaled <- return $ liftM show $ readExpr (args !! 0) >>= eval
    putStrLn $ extractValue $ trapError evaled
```

Here's what this new function is doing:

1. `args` is the list of command-line arguments;
2. `evaled` is the result of
   1. taking first argument (`args !!  0`)
   2. parsing it (`readExpr`)
   3. passing it to eval (`>>= eval`; the bind operation has higher precedence than function application)
   4. calling show on it within the Error monad.

   Note also that the whole action has type IO (Either LispError String), giving evaled type Either LispError String. It has to be, because our trapError function can only convert errors to Strings, and that type must match the type of normal values
3. caught is the result of
   1. calling `trapError` on `evaled`, converting errors to their string representation.
   2. calling `extractValue` to get a String out of this Either LispError String action.
   3. printing the results through `putStrLn`

Compile and run the new code, and try throwing it a couple errors:

```
$ ghc -package parsec -o errorcheck listing5.hs
$ ./errorcheck '(+ 2 "two")'
Invalid type: expected number, found "two"
$ ./errorcheck "(+ 2)"
Expected 2 args; found values 2
$ ./errorcheck "(what? 2)"
Unrecognized primitive function args: "what?"
```

6 Evaluation, Part 2

6.1 Additional Primitives: Partial Application

Now that we can deal with type errors, bad arguments, and so on, we'll flesh out our primitive list so that it does something more than calculate. We'll add boolean operators, conditionals, and some basic string operations.

Start by adding the following into the list of primitives:

```
("=", numBoolBinop (==)),
("<", numBoolBinop (<)),
(">", numBoolBinop (>)),
("/=", numBoolBinop (/=)),
(">=", numBoolBinop (>=)),
("<=", numBoolBinop (<=)),
("&&", boolBoolBinop (&&)),
("||", boolBoolBinop (||)),
("string=?", strBoolBinop (==)),
("string<?", strBoolBinop (<)),
("string>?", strBoolBinop (>)),
("string<=?", strBoolBinop (<=)),
("string>=?", strBoolBinop (>=)),
```

These depend on helper functions that we haven't written yet: `numBoolBinop` and `strBoolBinop`. Instead of taking a variable number of arguments and returning an integer, these both take exactly two arguments and return a boolean. They differ from each other only in the type of argument they expect, so let's factor the duplication into a generic `boolBinop` function that's parameteried by the unpacker function it applies to its arguments:

```
boolBinop :: (LispVal -> ThrowsError a) -> (a -> a -> Bool) -> [LispVal]
          -> ThrowsError LispVal

boolBinop unpacker op args = if length args /= 2
                             then throwError $ NumArgs 2 args
                             else do left <- unpacker $ args !! 0
                                     right <- unpacker $ args !! 1
                                     return $ Bool $ left ‘op‘ right
```

Because each arg may throw a type mismatch, we have to unpack them sequentially, in a do-block (for the Error monad). We then apply the operation to the two arguments and wrap the result in the Bool constructor. Any function can be turned into an infix operator by wrapping it in backticks (`‘op‘`).

Also, take a look at the type signature. `boolBinop` takes *two* functions as its first two arguments: the first is used to unpack the arguments from LispVals to native Haskell types, and the second is the actual operation to perform. By parameterizing different parts of the behavior, you make the function more reusable.

Now we define three functions that specialize boolBinop with different unpackers:

```
numBoolBinop  = boolBinop unpackNum
strBoolBinop  = boolBinop unpackStr
boolBoolBinop = boolBinop unpackBool
```

We haven't told Haskell how to unpack strings from LispVals yet. This works similarly to unpackNum, pattern matching against the value and either returning it or throwing an error. Again, if passed a primitive value that could be interpreted as a string (such as a number or boolean), it will silently convert it to the string representation.

```
unpackStr :: LispVal -> ThrowsError String
unpackStr (String s) = return s
unpackStr (Number s) = return $ show s
unpackStr (Bool s) = return $ show s
unpackStr notString = throwError $ TypeMismatch "string" notString
```

And we use similar code to unpack booleans:

```
unpackBool :: LispVal -> ThrowsError Bool
unpackBool (Bool b) = return b
unpackBool notBool = throwError $ TypeMismatch "boolean" notBool
```

Let's compile and test this to make sure it's working, before we proceed to the next feature:

```
$ ghc -package parsec -o simple_parser listing6.1.hs
$ ./simple_parser "(< 2 3)"
#t
$ ./simple_parser "(> 2 3)"
#f
$ ./simple_parser "(>= 3 3)"
#t
$ ./simple_parser '(string=? "test"  "test")'
#t
$ ./simple_parser '(string>? "abc" "bba")'
#t
```

6.2 Conditionals: Pattern Matching 2

Now, we'll proceed to adding an if-clause to our evaluator. As with standard Scheme, our evaluator considers `#f` to be false and any other value to be true:

```
eval (List [Atom "if", pred, conseq, alt]) =
    do result <- eval pred
       case result of
         Bool False -> eval alt
         otherwise -> eval conseq
```

This is another example of nested pattern-matching. Here, we're looking for a four-element list. The first element must be the atom `if`. The others can be any Scheme forms. We take the first element, evaluate, and if it's false, evaluate the alternative. Otherwise, we evaluate the consequent.

Compile and run this, and you'll be able to play around with conditionals:

```
$ ghc -package parsec -o simple_parser listing6.2.hs
$ ./simple_parser "(if (> 2 3) \"no\" \"yes\")"
"yes"
$ ./simple_parser "(if (= 3 3) (+ 2 3 (- 5 1)) \"unequal\")"
9
```

6.3 List Primitives: car, cdr, and cons

For good measure, lets also add in the basic list-handling primitives. Because we've chosen to represent our lists as Haskell algebraic data types instead of pairs, these are somewhat more complicated than their definitions in many Lisps. It's easiest to think of them in terms of their effect on printed S-expressions:

1. `(car (a b c)) = a`
2. `(car (a)) = a`
3. `(car (a b . c)) = a`
4. `(car a) = error (not a list)`
5. `(car a b) = error (car takes only one argument)`

We can translate these fairly straightforwardly into pattern clauses, recalling that `(x : xs)` divides a list into the first element and the rest:

```
car :: [LispVal] -> ThrowsError LispVal
car [List (x : xs)] = return x
car [DottedList (x : xs) _] = return x
car [badArg] = throwError $ TypeMismatch "pair" badArg
car badArgList = throwError $ NumArgs 1 badArgList
```

Let's do the same with cdr:

1. `(cdr (a b c)) = (b c)`
2. `(cdr (a b)) = (b)`
3. `(cdr (a)) = NIL`
4. `(cdr (a b . c)) = (b . c)`
5. `(cdr (a . b)) = b`
6. `(cdr a) = error (not list)`
7. `(cdr a b) = error (too many args)`

We can represent the first three cases with a single clause. Our parser represents `'()` as `List []`, and when you pattern-match `(x : xs)` against `[x]`, `xs` is bound to `[]`. The other ones translate to separate clauses:

```
cdr :: [LispVal] -> ThrowsError LispVal
cdr [List (x : xs)] = return $ List xs
cdr [DottedList (_ : xs) x] = return $ DottedList xs x
cdr [DottedList [xs] x] = return x
cdr [badArg] = throwError $ TypeMismatch "pair" badArg
cdr badArgList = throwError $ NumArgs 1 badArgList
```

Cons is a little tricky, enough that we should go through each clause case-by-case. If you cons together anything with `Nil`, you end up with a one-item list, the `Nil` serving as a terminator:

```
cons :: [LispVal] -> ThrowsError LispVal
cons [x1, List []] = return $ List [x1]
```

If you cons together anything and a list, it's like tacking that anything onto the front of the list:

```
cons [x, List xs] = return $ List $ [x] ++ xs
```

However, if the list is a DottedList, then it should stay a DottedList, taking into account the improper tail:

```
cons [x, DottedList xs xlast] = return $ DottedList ([x] ++ xs) xlast
```

If you cons together two non-lists, or put a list in front, you get a *DottedList*. This is because such a cons cell isn't terminated by the normal Nil that most lists are.

```
cons [x1, x2] = return $ DottedList [x1] x2
```

Finally, attempting to cons together more or less than 2 arguments is an error:

```
cons badArgList = throwError $ NumArgs 2 badArgList
```

Our last step is to implement eqv?. Scheme offers three levels of equivalence predicates: eq?, eqv?, and equal?. For our purposes, eq? and eqv? are basically the same: they recognize two items as the same if they print the same, and are fairly slow. So we can write one function for both of them and register it under eq? and eqv?.

```
eqv :: [LispVal] -> ThrowsError LispVal
eqv [(Bool arg1), (Bool arg2)] = return $ Bool $ arg1 == arg2
eqv [(Number arg1), (Number arg2)] = return $ Bool $ arg1 == arg2
eqv [(String arg1), (String arg2)] = return $ Bool $ arg1 == arg2
eqv [(Atom arg1), (Atom arg2)] = return $ Bool $ arg1 == arg2
eqv [(DottedList xs x), (DottedList ys y)] =
    eqv [List $ xs ++ [x], List $ ys ++ [y]]
eqv [(List arg1), (List arg2)] =
        return $ Bool $ (length arg1 == length arg2) &&
                        (and $ map eqvPair $ zip arg1 arg2)
    where eqvPair (x1, x2) = case eqv [x1, x2] of
                               Left err -> False
                               Right (Bool val) -> val
eqv [_, _] = return $ Bool False
eqv badArgList = throwError $ NumArgs 2 badArgList
```

Most of these clauses are self-explanatory, the exception being the one for two Lists. This, after checking to make sure the lists are the same length, zips the two lists of pairs, runs eqvPair on them to test if each corresponding pair is equal, and then uses the function and to return false if any of the resulting values is false. eqvPair is an example of a local definition: it is defined using the 'where' keyword, just like a normal function, but is available only within that particular clause of eqv.

Compile and run to try out the new list functions:

```
$ ghc -package parsec -o eqv listing6.3.hs
$ ./eqv "(car '(2 3 4))"
2
$ ./eqv "(cdr '(2 3 4))"
(3 4)
$ ./eqv "(car (cdr (cons 2 '(3 4))))"
3
```

6.4 Equal? and Weak Typing: Heterogenous Lists

Since we introduced weak typing above, we'd also like to introduce an equal? function that ignores differences in the type tags and only tests if two values can be interpreted the same. For example, `(eqv? 2 "2") = #f`, yet we'd like `(equal? 2 "2") = #t`. Basically, we want to try all of our unpack functions, and if any of them result in Haskell values that are equal, return true.

The obvious way to approach this is to store the unpacking functions in a list and use mapM to execute them in turn. Unfortunately, this doesn't work, because standard Haskell only lets you put objects in a list *if they're the same type*. The various unpacker functions return different types, so you can't store them in the same list.

We'll get around this by using a GHC extension - Existential Types - that lets us create a heterogenous list, subject to typeclass constraints. Extensions are fairly common in the Haskell world: they're basically necessary to create any reasonably large program, and they're often compatible between implementations (existential types work in both Hugs and GHC and are a candidate for standardization).

The first thing we need to do is define a data type that can hold any function from a LispVal to something, provided that that "something" supports equality:

```
data Unpacker = forall a. Eq a => AnyUnpacker (LispVal -> ThrowsError a)
```

This is like any normal algebraic datatype, except for the type constraint. It says, "For any type that is an instance of Eq, you can define an Unpacker that takes a function from LispVal to that type, and may throw an error". We'll have to wrap our functions with the AnyUnpacker constructor, but then we can create a list of Unpackers that does just what we want it.

Rather than jump straight to the equal? function, let's first define a helper function that takes an unpacker and then determines if two LispVals are equal when it unpacks them:

```
unpackEquals :: LispVal -> LispVal -> Unpacker -> ThrowsError Bool
unpackEquals arg1 arg2 (AnyUnpacker unpacker) =
             do unpacked1 <- unpacker arg1
                unpacked2 <- unpacker arg2
                return $ unpacked1 == unpacked2
        `catchError` (const $ return False)
```

After pattern-matching to retrieve the actual function, we enter a do-block for the ThrowsError monad. This retrieves the Haskell values of the two LispVals, and then tests whether they're equal. If there is an error anywhere within the two unpackers, it returns false, using the const function because catchError expects a function to apply to the error value.

Finally, we can define equal? in terms of these helpers:

```
equal :: [LispVal] -> ThrowsError LispVal
equal [arg1, arg2] = do
    primitiveEquals <- liftM or $ mapM (unpackEquals arg1 arg2)
                       [AnyUnpacker unpackNum,
                        AnyUnpacker unpackStr,
                        AnyUnpacker unpackBool]
    eqvEquals <- eqv [arg1, arg2]
    return $ Bool $ (primitiveEquals || let (Bool x) = eqvEquals in x)
equal badArgList = throwError $ NumArgs 2 badArgList
```

The first action makes a heterogenous list of [unpackNum, unpackStr, unpackBool], and then maps the partially-applied (unpackEquals arg1 arg2) over it. This gives a list of Bools, so we use the Prelude function or to return true if any single one of them is true.

The second action tests the two arguments with eqv?. Since we want equal? to be looser than eqv?, it should return true whenever eqv? does so. This also lets us avoid handling cases like the list or dotted-list (though this introduces a bug; see the second exercise in this section).

Finally, equal? ors both of these values together and wraps the result in the Bool constructor, returning a LispVal. The `let (Bool x) = eqvEquals in x` is a quick way of extracting a value from an algebraic type: it pattern matches Bool x against the eqvEquals value, and then returns x. The result of a let-expression is the expression following the keyword `in`.

To use these functions, insert them into our primitives list:

```
("car", car),
("cdr", cdr),
("cons", cons),
("eq?", eqv),
("eqv?", eqv),
("equal?", equal)]
```

To compile this code, you need to enable GHC extensions with `-fglasgow-exts`:

```
$ ghc -package parsec -fglasgow-exts -o parser listing6.4.hs
$ ./simple_parser "(cdr '(a simple test))"
(simple test)
$ ./simple_parser "(car (cdr '(a simple test)))"
simple
$ ./simple_parser "(car '((this is) a test))"
(this is)
$ ./simple_parser "(cons '(this is) 'test)"
```

```
((this is) . test)
$ ./simple_parser "(cons '(this is) '())"
((this is))
$ ./simple_parser "(eqv? 1 3)" #f
$ ./simple_parser "(eqv? 3 3)"
#t
$ ./simple_parser "(eqv? 'atom 'atom)"
#t
```

6.5 Exercises

1. Instead of treating any non-false value as true, change the definition of `if` so that the predicate accepts only Bool values and throws an error on any others.

2. equal? has a bug in that a list of values is compared using eqv? instead of equal?. For example, `(equal? '(1 "2") '(1 2)) = #f`, while you'd expect it to be true. Change equal? so that it continues to ignore types as it recurses into list structures. You can either do this explicitly, following the example in eqv?, or factor the list clause into a separate helper function that is parameterized by the equality testing function.

3. Implement cond and case expressions.

4. Add the rest of the string functions. You don't yet know enough to do string-set!; this is difficult to implement in Haskell, but you'll have enough information after the next 2 sections.

7 Building a REPL

So far, we've been content to evaluate single expressions from the command line, printing the result and exiting afterwards. This is fine for a calculator, but isn't what most people think of as "programming". We'd like to be able to define new functions and variables, and refer to them later. But before we can do this, we need to build a system that can execute multiple statements without exiting the program.

Instead of executing a whole program at once, we're going to build a *read-eval-print loop*. This reads in expressions from the console one at a time and executes them interactively, printing the result after each expression. Later expressions can reference variables set by earlier ones (or will be able to, after the next section), letting you build up libraries of functions.

First, we need to import some additional IO functions. Add the following to the top of the program:

```
import IO hiding (try)
```

We have to hide the try function (used in the IO module for exception handling) because we use Parsec's try function.

Next, we define a couple helper functions to simplify some of our IO tasks. We'll want a function that prints out a string and immediately flushes the stream; otherwise, output might sit in output buffers and the user will never see prompts or results.

```
flushStr :: String -> IO ()
flushStr str = putStr str >> hFlush stdout
```

Then, we create a function that prints out a prompt and reads in a line of input:

```
readPrompt :: String -> IO String
readPrompt prompt = flushStr prompt >> getLine
```

Pull the code to parse and evaluate a string and trap the errors out of main into its own function:

```
evalString :: String -> IO String
evalString expr =
    return $ extractValue $ trapError (liftM show $ readExpr expr >>= eval)
```

And write a function that evaluates a string and prints the result:

```
evalAndPrint :: String -> IO ()
evalAndPrint expr =  evalString expr >>= putStrLn
```

Now it's time to tie it all together. We want to read input, perform a function, and print the output, all in an infinite loop. The built-in function interact *almost* does what we want, but doesn't loop. If we used the combination `sequence . repeat . interact`, we'd get an infinite loop, but we wouldn't be able to break out of it. So we need to roll our own loop:

```
until_ :: Monad m => (a -> Bool) -> m a -> (a -> m ()) -> m ()
until_ pred prompt action = do
  result <- prompt
  if pred result
     then return ()
     else action result >> until_ pred prompt action
```

The underscore after the name is a typical naming convention in Haskell for monadic functions that repeat but do not return a value. `until_` takes a predicate that signals when to stop, an action to perform before the test, and a function-returning-an-action to do to the input. Each of the latter two is generalized over *any* monad, not just IO. That's why we write their types using the type variable `m`, and include the type constraint `Monad m =>`.

Note also that we can write recursive actions just as we write recursive functions.

Now that we have all the machinery in place, we can write our REPL easily:

```
runRepl :: IO ()
runRepl = until_ (== "quit") (readPrompt "Lisp>>> ") evalAndPrint
```

And change our main function so it either executes a single expression, or enters the REPL and continues evaluating expressions until we type `quit`:

```
main :: IO ()
main = do args <- getArgs
          case length args of
              0 -> runRepl
              1 -> evalAndPrint $ args !! 0
              otherwise -> putStrLn "Program takes only 0 or 1 argument"
```

Compile and run the program, and try it out:

```
$ ghc -package parsec -fglasgow-exts -o lisp listing7.hs
$ ./lisp
Lisp>>> (+ 2 3)
5
Lisp>>> (cons this '())
Unrecognized special form: this
Lisp>>> (cons 2 3)
(2 . 3)
Lisp>>> (cons 'this '())
(this)
Lisp>>> quit
$
```

8 Adding Variables: State in Haskell

Finally, we get to the good stuff: variables. A variable lets us save the result of an expression and refer to it later. In Scheme, a variable can also be reset to new values, so that its value changes as the program executes. This presents a complication for Haskell, because the execution model is built upon functions that return values, but never change them.

Nevertheless, there are several ways to simulate state in Haskell, all involving monads. The simplest is probably the State monad, which lets you hide arbitrary state within the monad and pass it around behind the scenes. You specify the state type as a parameter to the monad (eg. if a function returns an integer but modifies a list of string pairs, it would have type `State [(String, String)] Integer`), and access it via the get and put functions, usually within a do-block. You'd specify the initial state via the `runState myStateAction initialList`, which returns a pair containing the return value and the final state.

Unfortunately, the state monad doesn't work well for us, because the type of data we need to store is fairly complex. For a simple top-level environment, we could get away with `[(String, LispVal)]`, storing mappings from variable names to values. However, when we start dealing with function calls, these mappings become a stack of nested environments, arbitrarily deep. And when we add closures, environments might get saved in an arbitrary Function value, and passed around throughout the program. In fact, they might be saved in a variable and passed out of the runState monad entirely, something we're not allowed to do.

Instead, we use a feature called *state threads*, letting Haskell manage the aggregate state for us. This lets us treat mutable variables as we would in any other programming language, using functions to get or set variables. There are two flavors of state threads: the ST monad creates a stateful computation that can be executed as a unit, without the state escaping to the rest of the program. The IORef module lets you use stateful variables within the IO monad. Since our state has to be interleaved with IO anyway (it persists between lines in the REPL, and we will eventually have IO functions within the language itself), we'll be using IORefs.

We can start out by importing Data.IORef and defining a type for our environments:

```
import Data.IORef

type Env = IORef [(String, IORef LispVal)]
```

This declares an Env as an IORef holding a list that maps Strings to mutable LispVals. We need IORefs for both the list itself and for individual values because there are *two* ways that the program can mutate the environment. It might use `set!` to change the value of an individual variable, a change visible to any function that shares that environment (Scheme allows nested scopes, so a variable in an outer scope is visible to all inner scopes). Or it might use `define` to add a new variable, which should be visible on all subsequent statements.

Since IORefs can only be used within the IO monad, we'll want a helper action to create an empty environment. We can't just use the empty list `[]` because all accesses to IORefs must be sequenced, and so the type of our null environment is IO Env instead of just plain Env:

```
nullEnv :: IO Env
nullEnv = newIORef []
```

From here, things get a bit more complicated, because we'll be simultaneously dealing with *two* monads. Remember, we also need an Error monad to handle things like unbound variables. The parts that need IO functionality and the parts that may throw exceptions are interleaved, so we can't just catch all the exceptions and return only normal values to the IO monad.

Haskell provides a mechanism known as *monad transformers* that lets you combine the functionality of multiple monads. We'll be using one of these - ErrorT - which lets us layer error-handling functionality on top of the IO monad. Our first step is create a type synonym for our combined monad:

```
type IOThrowsError = ErrorT LispError IO
```

Like IOThrows, IOThrowsError is really a type constructor: we've left off the last argument, the return type of the function. However, ErrorT takes one more argument than plain old Either: we have to specify the type of monad that we're layering our error-handling functionality over. We've created a monad that may contain IO actions that throw a LispError.

We have a mix of IOThrows and IOThrowsError functions, but actions of different types cannot be contained within the same do-block, even if they provide essentially the same functionality. Haskell already provides a mechanism - lifting to bring values of the lower type (IO) into the combined monad. Unfortunately, there's no similar support to bring a value of the untransformed upper type into the combined monad, so we need to write it ourselves:

```
liftThrows :: ThrowsError a -> IOThrowsError a
liftThrows (Left err) = throwError err
liftThrows (Right val) = return val
```

This destructures the Either type and either re-throws the error type or returns the ordinary value. Methods in typeclasses resolve based on the type of the expression, so throwError and return (members of MonadError and Monad, respectively) take on their IOThrowsError definitions. Incidentally, the type signature provided here is not fully general: if we'd left it off, the compiler would have inferred `liftThrows :: MonadError m => Either e a -> m e`.

We'll also want a helper function to run the whole top-level IOThrowsError action, returning an IO action. We can't escape from the IO monad, because a function that performs IO has an effect on the outside world, and you don't want that in a lazily-evaluated pure function. But you can run the error computation and catch the errors.

```
runIOThrows :: IOThrowsError String -> IO String
runIOThrows action = runErrorT (trapError action) >>= return . extractValue
```

This uses our previously-defined `trapError` function to take any error values and convert them to their string representations, then runs the whole computation via runErrorT. The result is passed into extractValue and returned as a value in the IO monad.

Now we're ready to return to environment handling. We'll start with a function to determine if a given variable is already bound in the environment, necessary for proper handling of define:

```
isBound :: Env -> String -> IO Bool
isBound envRef var =
    readIORef envRef >>= return . maybe False (const True) . lookup var
```

This first extracts the actual environment value from its IORef via readIORef. Then we pass it to lookup to search for the particular variable we're interested in. lookup returns a Maybe value, so we return False if that value was Nothing and True otherwise (we need to use the const function because maybe expects a function to perform on the result and not just a value). Finally, we use return to lift that value into the IO monad. Since we're just interested in a true/false value, we don't need to deal with the actual IORef that lookup returns.

Next, we'll want to define a function to retrieve the current value of a variable:

```
getVar :: Env -> String -> IOThrowsError LispVal
getVar envRef var =
    do env <- liftIO $ readIORef envRef
       maybe (throwError $ UnboundVar "Getting an unbound variable" var)
             (liftIO . readIORef)
             (lookup var env)
```

Like the previous function, this begins by retrieving the actual environment from the IORef. However, getVar uses the IOThrowsError monad, because it also needs to do some error handling. As a result, we need to use the liftIO function to lift the readIORef action into the combined monad. Similarly, when we return the value, we use liftIO . readIORef to generate an IOThrowsError action that reads the returned IORef. We don't need to use liftIO to throw an error, however, because throwError is a defined for the MonadError typeclass, of which ErrorT is an instance.

Now we create a function to set values:

```
setVar :: Env -> String -> LispVal -> IOThrowsError LispVal
setVar envRef var value =
    do env <- liftIO $ readIORef envRef
       maybe (throwError $ UnboundVar "Setting an unbound variable" var)
             (liftIO . (flip writeIORef value))
             (lookup var env)
       return value
```

Again, we first read the environment out of its IORef and run a lookup on it. This time, however, we want to change the variable instead of just reading it. The writeIORef action provides a means for this, but takes its arguments in the wrong order (`ref -> value` instead of `value -> ref`). So we use the built-in function flip to switch the arguments of writeIORef around, and then pass it the value. Finally, we return the value we just set, for convenience.

We'll want a function to handle the special behavior of define, which sets a variable if already bound or creates a new one if not. Since we've already defined a function to set values, we can use it in the former case:

```
defineVar :: Env -> String -> LispVal -> IOThrowsError LispVal
defineVar envRef var value = do
    alreadyDefined <- liftIO $ isBound envRef var
    if alreadyDefined
       then setVar envRef var value >> return value
       else liftIO $ do
          valueRef <- newIORef value
          env <- readIORef envRef
          writeIORef envRef ((var, valueRef) : env)
          return value
```

It's the latter case that's interesting, where the variable is unbound. We create an IO action (via do-notation) that creates a new IORef to hold the new variable, reads the current value of the environment, then writes a new list back to that variable consisting of the new (key, variable) pair added to the front of the list. Then we lift that whole do-block into the IOThrowsError monad with liftIO.

There's one more useful environment function: being able to bind a whole bunch of variables at once, like what would happen at function invocation. We might as well build that functionality now, though we won't be using it until the next section:

```
bindVars :: Env -> [(String, LispVal)] -> IO Env
bindVars envRef bindings = readIORef envRef >>= extendEnv bindings >>= newIORef
    where extendEnv bindings env = liftM (++ env) (mapM addBinding bindings)
          addBinding (var, value) = do ref <- newIORef value
                                       return (var, ref)
```

This is perhaps more complicated than the other functions, since it uses a monadic pipeline (rather than do-notation) and a pair of helper functions to do the work. It's best to start with the helper functions. addBinding takes a variable name and value, creates an IORef to hold the new variable , and then returns the (name, value) pair. extendEnv calls addBinding on each member of bindings (mapM) to create a list of (String, IORef LispVal) pairs, and then appends the current environment to the end of that `(++ env)`. Finally, the whole function wires these functions up in a pipeline, starting by reading the existing environment out of its IORef, then passing the result to extendEnv, then returning a new IORef with the extended environment.

Now that we have all our environment functions, we need to start using them in the evaluator. Since Haskell has no global variables, we'll have to thread the environment through the evaluator as a parameter. While we're at it, we might as well add the set! and define special forms.

```
eval :: Env -> LispVal -> IOThrowsError LispVal
eval env val@(String _) = return val
eval env val@(Number _) = return val
eval env val@(Bool _) = return val
eval env (Atom id) = getVar env id
eval env (List [Atom "quote", val]) = return val
eval env (List [Atom "if", pred, conseq, alt]) =
    do result <- eval env pred
       case result of
         Bool False -> eval env alt
         otherwise -> eval env conseq
eval env (List [Atom "set!", Atom var, form]) =
    eval env form >>= setVar env var
eval env (List [Atom "define", Atom var, form]) =
    eval env form >>= defineVar env var
eval env (List (Atom func : args)) =
    mapM (eval env) args >>= liftThrows . apply func
eval env badForm =
    throwError $ BadSpecialForm "Unrecognized special form" badForm
```

Since a single environment gets threaded through a whole interactive session, we need to change a few of our IO functions to take an environment.

```
evalAndPrint :: Env -> String -> IO ()
evalAndPrint env expr =  evalString env expr >>= putStrLn

evalString :: Env -> String -> IO String
evalString env expr =
    runIOThrows $ liftM show $ (liftThrows $ readExpr expr) >>= eval env
```

We need the runIOThrows in evalString because the type of the monad has changed from ThrowsError to IOThrowsError. Similarly, we need a liftThrows to bring readExpr into the IOThrowsError monad.

Next, we initialize the environment with a null variable before starting the program:

```
runOne :: String -> IO ()
runOne expr = nullEnv >>= flip evalAndPrint expr

runRepl :: IO ()
runRepl = nullEnv >>= until_ (== "quit") (readPrompt "Lisp>>> ") . evalAndPrint
```

We've created an additional helper function runOne to handle the single-expression case, since it's now somewhat more involved than just running evalAndPrint. The changes to runRepl are a bit more subtle: notice how we added a function composition operator before evalAndPrint. That's because evalAndPrint now takes an additional Env parameter, fed from nullEnv. The function composition tells `until_` that instead of taking plain old evalAndPrint as an action, it ought to apply it first to whatever's coming down the monadic pipeline, in this case the result of nullEnv. Thus, the actual function that gets applied to each line of input is (evalAndPrint env), just as we want it.

Finally, we need to change our main function to call runOne instead of evaluating evalAndPrint directly:

```
main :: IO ()
main = do args <- getArgs
          case length args of
              0 -> runRepl
              1 -> runOne $ args !! 0
              otherwise -> putStrLn "Program takes only 0 or 1 argument"
```

And we can compile and test our program:

```
$ ghc -package parsec -o lisp listing8.hs
$ ./lisp
Lisp>>> (define x 3)
3
Lisp>>> (+ x 2)
5
Lisp>>> (+ y 2)
Getting an unbound variable: y
Lisp>>> (define y 5)
5
Lisp>>> (+ x (- y 2))
6
Lisp>>> (define str "A string")
"A string"
Lisp>>> (< str "The string")
Invalid type: expected number, found "A string"
Lisp>>> (string<? str "The string")
#t
```

9 Defining Scheme Functions

Now that we can define variables, we might as well extend it to functions. After this section, you'll be able to define your own functions within Scheme and use them from other functions. Our implementation is nearly finished.

Let's start by defining new LispVal constructors:

```
| PrimitiveFunc ([LispVal] -> ThrowsError LispVal)
| Func {params :: [String], vararg :: (Maybe String),
        body :: [LispVal], closure :: Env}
```

We've added a separate constructor for primitives, because we'd like to be able to store `+`, `eqv?`, etc. in variables and pass them to functions. The PrimitiveFunc constructor stores a function that takes a list of arguments to a ThrowsError LispVal, the same type that is stored in our primitive list.

We also want a constructor to store user-defined functions. We store 4 pieces of information:

1. the names of the parameters, as they're bound in the function body;
2. whether the function accepts a variable-length list of arguments, and if so, the variable name it's bound to;
3. the function body, as a list of expressions; and
4. the environment that the function was created in.

This is an example of a record type. Records are somewhat clumsy in Haskell, so we're only using them for demonstration purposes. However, they can be invaluable in large-scale programming.

Next, we'll want to edit our `show` function to include the new types:

```
showVal (PrimitiveFunc _) = "<primitive>"
showVal (Func {params = args, vararg = varargs, body = body, closure = env}) =
  "(lambda (" ++ unwords (map show args) ++
     (case varargs of
        Nothing -> ""
        Just arg -> " . " ++ arg) ++ ") ...)"
```

Instead of showing the full function, we just print out the word "<primitive>" for primitives and the header info for user-defined functions. This is an example of pattern-matching for records: as with normal algebraic types, a pattern looks exactly like a constructor call. Field names come first and the variables they'll be bound to come afterwards.

Next, we need to change apply. Instead of being passed the name of a function, it'll be passed a LispVal representing the actual function. For primitives, that makes the code simpler: we need only read the function out of the value and apply it.

```
apply :: LispVal -> [LispVal] -> IOThrowsError LispVal
apply (PrimitiveFunc func) args = liftThrows $ func args
```

The interesting code happens when we're faced with a user defined function. Records let you pattern match on both the field name (as shown above) or the field position, so we'll use the latter form:

```
apply (Func params varargs body closure) args =
    if num params /= num args && varargs == Nothing
       then throwError $ NumArgs (num params) args
       else (liftIO $ bindVars closure $ zip params args) >>=
            bindVarArgs varargs >>= evalBody
    where remainingArgs = drop (length params) args
          num = toInteger . length
          evalBody env = liftM last $ mapM (eval env) body
          bindVarArgs arg env = case arg of
              Just argName -> liftIO $ bindVars env [(argName,
                                                       List $ remainingArgs)]
              Nothing -> return env
```

The very first thing this function does is check the length of the parameter list against the expected number of arguments. It throws an error if they don't match. We define a local function num to enhance readability and make the program a bit shorter.

Assuming the call is valid, we do the bulk of the call in monadic pipeline that binds the arguments to a new environment and executes the statements in the body. The first thing we do is zip the list of parameter names and the list of (already-evaluated) argument values together into a list of pairs. Then, we take that and the function's closure (*not* the current environment - this is what gives us lexical scoping) and use them to create a new environment to evaluate the function in. The result is of type IO, while the function as a whole is IOThrowsError, so we need to liftIO it into the combined monad.

Now it's time to bind the remaining arguments to the varArgs variable, using the local function bindVarArgs. If the function doesn't take varArgs (the Nothing clause), then we just return the existing environment. Otherwise, we create a singleton list that has the variable name as the key and the remaining args as the value, and pass that to bindVars. We define the local variable remainingArgs for readability, using the built-in function drop to ignore all the arguments that have already been bound to variables.

The final stage is to evaluate the body in this new environment. We use the local function evalBody for this, which maps the monadic function eval env over every statement in the body, and then returns the value of the last statement.

Since we're now storing primitives as regular values in variables, we have to bind them when the program starts up:

```
primitiveBindings :: IO Env
primitiveBindings =
        nullEnv >>= (flip bindVars $ map makePrimitiveFunc primitives)
    where makePrimitiveFunc (var, func) = (var, PrimitiveFunc func)
```

This takes the initial null environment, makes a bunch of name/value pairs consisting of PrimitiveFunc wrappers, and then binds the new pairs into the new environment. We also want to change `runOne` and `runRepl` to primitiveBindings instead:

```
runOne :: String -> IO ()
runOne expr = primitiveBindings >>= flip evalAndPrint expr

runRepl :: IO ()
runRepl = primitiveBindings >>=
          until_ (== "quit") (readPrompt "Lisp>>> ") . evalAndPrint
```

Finally, we need to change the evaluator to support lambda and function define. We'll start by creating a few helper functions to make it a little easier to create function objects in the IOThrowsError monad:

```
makeFunc varargs env params body =
    return $ Func (map showVal params) varargs body env
makeNormalFunc = makeFunc Nothing
makeVarargs = makeFunc . Just . showVal
```

Here, makeNormalFunc and makeVarArgs should just be considered specializations of makeFunc with the first argument set appropriately for normal functions vs. variable args. This is a good example of how to use first-class functions to simplify code.

Now, we can use them to add our extra eval clauses. They should be inserted after the define-variable clause and before the function-application one:

```
eval env (List (Atom "define" : List (Atom var : params) : body)) =
    makeNormalFunc env params body >>= defineVar env var
eval env (List (Atom "define" : DottedList (Atom var : params) varargs : body)) =
    makeVarargs varargs env params body >>= defineVar env var
eval env (List (Atom "lambda" : List params : body)) =
    makeNormalFunc env params body
eval env (List (Atom "lambda" : DottedList params varargs : body)) =
    makeVarargs varargs env params body
eval env (List (Atom "lambda" : varargs@(Atom _) : body)) =
    makeVarargs varargs env [] body
eval env (List (function : args)) = do
    func <- eval env function
    argVals <- mapM (eval env) args
    apply func argVals
```

As you can see, they just use pattern matching to destructure the form and then call the appropriate function helper. In define's case, we also feed the output into defineVar to bind a variable in the local environment. We also need to change the function application clause to remove the liftThrows function, since `apply` now works in the IOThrowsError monad.

We can now compile and run our program, and use it to write real programs!

```
$ ghc -package parsec -fglasgow-exts -o lisp listing9.hs
$ ./lisp
Lisp>>> (define (f x y) (+ x y))
(lambda ("x" "y") ...)
```

```
Lisp>>> (f 1 2)
3
Lisp>>> (f 1 2 3)
Expected 2 args; found values 1 2 3
Lisp>>> (f 1)
Expected 2 args; found values 1
Lisp>>> (define (factorial x) (if (= x 1) 1 (* x (factorial (- x 1)))))
(lambda ("x") ...)
Lisp>>> (factorial 10)
3628800
Lisp>>> (define (counter inc) (lambda (x) (set! inc (+ x inc)) inc))
(lambda ("inc") ...)
Lisp>>> (define my-count (counter 5))
(lambda ("x") ...)
Lisp>>> (my-count 3)
8
Lisp>>> (my-count 6)
14
Lisp>>> (my-count 5)
19
```

10 Creating IO primitives

Our Scheme can't really communicate with the outside world yet, so it would be nice if we could give it some I/O functions. Also, it gets really tedious typing in functions every time we start the interpreter, so it would be nice to load files of code and execute them.

The first thing we'll need is a new constructor for LispVals. PrimitiveFuncs have a specific type signature that doesn't include the IO monad, so they can't perform any IO. We want a dedicated constructor for primitive functions that perform IO:

```
| IOFunc ([LispVal] -> IOThrowsError LispVal)
```

While we're at it, let's also define a constructor for the Scheme data type of a port. Most of our IO functions will take one of these to read from or write to:

```
| Port Handle
```

A Handle is basically the Haskell notion of a port: it's an opaque data type, returned from openFile and similar IO actions, that you can read and write to.

For completeness, we ought to provide showVal methods for the new data types:

```
showVal (Port _) = "<IO port>"
showVal (IOFunc _) = "<IO primitive>"
```

This'll let the REPL function properly and not crash when you use a function that returns a port.

We'll need to make some minor changes to our parser to support load. Since Scheme files usually contain several definitions, we need to add a parser that will support several expressions, separated by whitespace. And it also needs to handle errors. We can re-use much of the existing infrastructure by factoring our basic readExpr so that it takes the actual parser as a parameter:

```
readOrThrow :: Parser a -> String -> ThrowsError a
readOrThrow parser input = case parse parser "lisp" input of
    Left err -> throwError $ Parser err
    Right val -> return val
```

```
readExpr = readOrThrow parseExpr
readExprList = readOrThrow (endBy parseExpr spaces)
```

Again, think of both readExpr and readExprList as specializations of the newly-renamed readOrThrow. We'll be using readExpr in our REPL to read single expressions; we'll be using readExprList from within load to read programs.

Next, we'll want a new list of IO primitives, structured just like the existing primitive list:

```
ioPrimitives :: [(String, [LispVal] -> IOThrowsError LispVal)]
ioPrimitives = [("apply", applyProc),
                ("open-input-file", makePort ReadMode),
                ("open-output-file", makePort WriteMode),
                ("close-input-port", closePort),
                ("close-output-port", closePort),
                ("read", readProc),
                ("write", writeProc),
                ("read-contents", readContents),
                ("read-all", readAll)]
```

The only difference here is in the type signature. Unfortunately, we can't use the existing primitive list because lists cannot contain elements of different types. We also need to change the definition of primitiveBindings to add our new primitives:

```
primitiveBindings :: IO Env
primitiveBindings =
        nullEnv >>= (flip bindVars $ map (makeFunc IOFunc) ioPrimitives
                                  ++ map (makeFunc PrimitiveFunc) primitives)
    where makeFunc constructor (var, func) = (var, constructor func)
```

We've generalized makeFunc to take a constructor argument, and now call it on the list of ioPrimitives in addition to the plain old primitives.

Now we start defining the actual functions. applyProc is a very thin wrapper around apply, responsible for destructuring the argument list into the form apply expects:

```
applyProc :: [LispVal] -> IOThrowsError LispVal
applyProc [func, List args] = apply func args
applyProc (func : args) = apply func args
```

makePort wraps the Haskell function openFile, converting it to the right type and wrapping its return value in the Port constructor. It's intended to be partially-applied to the IOMode, ReadMode for open-input-file and WriteMode for open-output-file:

```
makePort :: IOMode -> [LispVal] -> IOThrowsError LispVal
makePort mode [String filename] = liftM Port $ liftIO $ openFile filename mode
```

closePort also wraps the equivalent Haskell procedure, this time hClose:

```
closePort :: [LispVal] -> IOThrowsError LispVal
closePort [Port port] = liftIO $ hClose port >> (return $ Bool True)
closePort _ = return $ Bool False
```

readProc (named to avoid a name conflict with the built-in read) wraps the Haskell hGetLine and then sends the result to parseExpr, to be turned into a LispVal suitable for Scheme:

```
readProc :: [LispVal] -> IOThrowsError LispVal
readProc [] = readProc [Port stdin]
readProc [Port port] = (liftIO $ hGetLine stdin) >>= liftThrows . readExpr
```

Notice how hGetLine is of type `IO String` yet readExpr is of type `String -> ThrowsError LispVal`, so they both need to be converted (with liftIO and liftThrows, respectively) to the IOThrowsError monad. Only then can they be piped together with the monadic bind operator.

writeProc converts a LispVal to a string and then writes it out on the specified port:

```
writeProc :: [LispVal] -> IOThrowsError LispVal
writeProc [obj] = writeProc [obj, Port stdout]
writeProc [obj, Port port] = liftIO $ hPrint port obj >> (return $ Bool True)
```

We don't have to explicitly call show on the object we're printing, because hPrint takes a value of type Show a. It's calling show for us automatically. This is why we bothered making LispVal an instance of Show; otherwise, we wouldn't be able to use this automatic conversion and would have to call showVal ourselves. Many other Haskell functions also take instances of Show, so if we'd extended this with other IO primitives, it could save us significant labor.

readContents reads the whole file into a string in memory. It's a thin wrapper around Haskell's readFile, again just lifting the IO action into an IOThrowsError action and wrapping it in a String constructor:

```
readContents :: [LispVal] -> IOThrowsError LispVal
readContents [String filename] = liftM String $ liftIO $ readFile filename
```

The helper function `load` doesn't do what Scheme's load does (we handle that later). Rather, it's responsible only for reading and parsing a file full of statements. It's used in two places: readAll (which returns a list of values) and load (which evaluates those values as Scheme expressions).

```
load :: String -> IOThrowsError [LispVal]
load filename = (liftIO $ readFile filename) >>= liftThrows . readExprList
```

readAll then just wraps that return value with the List constructor:

```
readAll :: [LispVal] -> IOThrowsError LispVal
readAll [String filename] = liftM List $ load filename
```

Implementing the actual Scheme load function is a little tricky, because load can introduce bindings into the local environment. Apply, however, doesn't take an environment argument, and so there's no way for a primitive function (or any function) to do this. We get around this by implementing load as a special form:

```
eval env (List [Atom "load", String filename]) =
    load filename >>= liftM last . mapM (eval env)
```

Finally, we might as well change our runOne function so that instead of evaluating a single expression from the command line, it takes the name of a file to execute and runs that as a program. Additional command-line arguments will get bound into a list `args` within the Scheme program:

```
runOne :: [String] -> IO ()
runOne args = do
    env <-
primitiveBindings >>= flip bindVars [("args", List $ map String $ drop 1 args)]
    (runIOThrows $ liftM show $ eval env (List [Atom "load", String (args !! 0)]))
        >>= hPutStrLn stderr
```

That's a little involved, so let's go through it step-by-step. The first line takes the original primitive bindings, passes that into bindVars, and then adds a variable named `args` that's bound to a List containing String versions of all but the first argument. (The first argument is the filename to execute.) Then, it creates a Scheme form (load `arg1`), just as if the user had typed it in, and evaluates it. The result is transformed to a string (remember, we have to do this before catching errors, because the error handler converts them to strings and the types must match) and then we run the whole IOThrowsError action. Then we print the result on STDERR. (Traditional UNIX conventions hold that STDOUT should be used only or program output, with any error messages going to stderr. In this case, we'll also be printing the return value of the last statement in the program, which generally has no meaning to anything.)

Then we change main so it uses our new runOne function. Since we no longer need a third clause to handle the wrong number of command-line arguments, we can simplify it to an if statement:

```
main :: IO ()
main = do args <- getArgs
          if null args then runRepl else runOne $ args
```

11 Towards a Standard Library

Our Scheme is almost complete now, but it's still rather hard to use. At the very least, we'd like a library of standard list-manipulation functions that we can use to perform some common computations.

Rather than using a typical Scheme implementation, defining each list function in terms of a recursion on lists, we'll implement two primitive recursion operators (`fold` and `unfold`) and then define our whole library based on those. This style is used by the Haskell Prelude: it gives you more concise definitions, less room for error, and good practice using fold to capture iteration.

We'll start by defining a few obvious helper functions. `not` and `null` are defined exactly as you'd expect it, using if statements:

```
(define (not x)            (if x #f #t))
(define (null? obj)        (if (eqv? obj '()) #t #f))
```

We can use the varargs feature to define list, which just returns a list of its arguments:

```
(define (list . objs)       objs)
```

We also want an `id` function, which just returns its argument unchanged. This may seem completely useless - if you already have a value, why do you need a function to return it? However, several of our algorithms expect a function that tells us what to do with a given value. By defining `id`, we let those higher-order functions work even if we don't want to do *anything* with the value.

```
(define (id obj)           obj)
```

Similarly, it'd be nice to have a `flip` function, in case we want to pass in a function that takes its arguments in the wrong order:

```
(define (flip func)        (lambda (arg1 arg2) (func arg2 arg1)))
```

Finally, we add `curry` and `compose`, which work like their Haskell equivalents (partial-application and the dot operator, respectively).

```
(define (curry func arg1)  (lambda (arg) (apply func (cons arg1 arg))))
(define (compose f g)      (lambda (arg) (f (apply g arg))))
```

We might as well define some simple library functions that appear in the Scheme standard:

```
(define zero?              (curry = 0))
(define positive?          (curry < 0))
(define negative?          (curry > 0))
(define (odd? num)         (= (mod num 2) 1))
(define (even? num)        (= (mod num 2) 0))
```

These are basically done just as you'd expect them. Note the usage of curry to define zero?, positive? and negative?. We bind the variable zero? to the function returned by curry, giving us an unary function that returns true if its argument is equal to zero.

Next, we want to define a fold function that captures the basic pattern of recursion over a list. The best way to think about fold is to picture a list in terms of its infix constructors: `[1, 2, 3, 4] = 1:2:3:4:[]` in Haskell or `(1 . (2 . (3 . (4 . NIL))))` in Scheme. A fold function replaces every constructor with a binary operation, and replaces NIL with the accumulator. So, for example, (fold + 0 '(1 2 3 4)) = (1 + (2 + (3 + (4 + 0)))).

With that definition, we can write our fold function. Start with a right-associative version to mimic the above examples:

```
(define (foldr func end lst)
  (if (null? lst)
      end
      (func (car lst) (foldr func end (cdr lst)))))
```

The structure of this function mimics our definition almost exactly. If the list is null, replace it with the end value. If not, apply the function to the car of the list and to the result of folding this function and end value down the rest of the list. Since the right-hand operand is folded up first, you end up with a right-associative fold.

We also want a left-associative version. For most associative operations like + and *, the two of them are completely equivalent. However, there is at least one important binary operation that is *not* associative: cons. For all our list manipulation functions, then, we'll need to deliberately choose between left- and right-associative folds.

```
(define (foldl func accum lst)
  (if (null? lst)
      accum
      (foldl func (func accum (car lst)) (cdr lst))))
```

This begins the same way as the right-associative version, with the test for null that returns the accumulator. This time, however, we apply the function to the accumulator and first element of the list, instead of applying it to the first element and the result of folding the list. This means that we process the beginning first, giving us left-associativity. Once we reach the end of the list, '(), we then return the result that we've been progressively building up.

Note that func takes its arguments in the opposite order from foldr. In foldr, the accumulator represents the *rightmost* value to tack onto the end of the list, after you've finished recursing down it. In foldl, it represents the completed calculation for the *leftmost* part of the list. In order to preserve our intuitions about commutativity of operators, it should therefore be the left argument of our operation in foldl, but the right argument in foldr.

Once we've got our basic folds, we can define a couple convenience names to match typical Scheme usage:

```
(define fold foldl)
(define reduce fold)
```

These are just new variables bound to the existing functions: they don't define new functions. Most Schemes call fold `reduce` or plain old `fold`, and don't make the distinction between foldl and foldr. We define it to be foldl, which happens to be tail-recursive and hence runs more efficiently than foldr (it doesn't have to recurse all the way down to the end of the list before it starts building up the computation). Not all operations are associative, however; we'll see some cases later where we *have* to use foldr to get the right result.

Next, we want to define a function that is the opposite of fold. Given an unary function, an initial value, and a unary predicate, it continues applying the function to the last value until the predicate is true, building up a list as it goes along. This is essentially what generators are in Python or Icon:

```
(define (unfold func init pred)
  (if (pred init)
      (cons init '())
      (cons init (unfold func (func init) pred))))
```

As usual, our function structure basically matches the definition. If the predicate is true, then we cons a `'()` onto the last value, terminating the list. Otherwise, cons the result of unfolding the next value (func init) onto the current value.

In academic functional programming literature, folds are often called *catamorphisms*, unfolds are often called *anamorphisms*, and the combinations of the two are often called *hylomorphisms.* They're interesting because *any for-each loop can be represented as a catamorphism.* To convert from a loop to a foldl, package up all mutable variables in the loop into a data structure (records work well for this, but you can also use an algebraic data type or a list). The initial state becomes the accumulator; the loop body becomes a function with the loop variables as its first argument and the iteration variable as its second; and the list becomes, well, the list. The result of the fold function is the new state of all the mutable variables.

Similarly, *every for-loop (without early exits) can be represented as a hylomorphism.* The initialization, termination, and step conditions of a for-loop define an anamorphism that builds up a list of values for the iteration variable to take. Then, you can treat that as a for-each loop and use a catamorphism to break it down into whatever state you wish to modify.

Let's go through a couple examples. We'll start with typical sum/product/and/or functions:

```
(define (sum . lst)         (fold + 0 lst))
(define (product . lst)     (fold * 1 lst))
(define (and . lst)         (fold && #t lst))
(define (or . lst)          (fold || #f lst))
```

These all follow from the definitions:

1.
   ```
   (sum 1 2 3 4)
   = 1 + 2 + 3 + 4 + 0
   = (fold + 0 '(1 . (2 . (3 . (4 . NIL)))))
   ```

2.
   ```
   (product 1 2 3 4)
   = 1 * 2 * 3 * 4 * 1
   = (fold * 1 '(1 . (2 . (3 . (4 . NIL)))))
   ```

3.
   ```
   (and #t #t #f)
   = #t && #t && #f && #t
   = (fold && #t '(#t . (#t . (#f . NIL))))
   ```

4.
   ```
   (or #t #t #f)
   = #t || #t || #f || #f
   = (fold  || #f '(#t . (#t . (#f . NIL))))
   ```

Since all of these operators are associative, it doesn't matter whether we use foldr or foldl. We replace the cons constructor with the operator, and the nil constructor with the identity element for that operator.

Next, let's try some more complicated operators. Max and min find the maximum and minimum of their arguments, respectively:

```
(define (max first . num-list)
  (fold (lambda (old new) (if (> old new) old new))
        first
        num-list))
```

```
(define (min first . num-list)
  (fold (lambda (old new) (if (< old new) old new))
        first
        num-list))
```

It's not immediately obvious what operation to fold over the list, because none of the built-ins quite qualify. Instead, think back to fold as a representation of a foreach loop. The accumulator represents any state we've maintained over previous iterations of the loop, so we'll want it to be the maximum value we've found so far. That gives us our initialization value: we want to start off with the leftmost variable in the list (since we're doing a foldl). Now recall that the result of the operation becomes the new accumulator at each step, and we've got our function. If the previous value is greater, keep it. If the new value is greater, or they're equal, return the new value. Reverse the operation for `min`.

How about `length`? We know that we can find the length of a list by counting down it, but how do we translate that into a fold?

```
(define (length lst)
  (fold (lambda (x y) (+ x 1)) 0 lst))
```

Again, think in terms of its definition as a loop. The accumulator starts off at 0 and gets incremented by 1 with each iteration. That gives us both our initialization value - 0 - and our function - `(lambda (x y) (+ x 1))`. Another way to look at this is "The length of a list is 1 + the length of the sublist to its left".

Let's try something a bit trickier: `reverse`.

```
(define (reverse lst)
  (fold (flip cons) '() lst))
```

The function here is fairly obvious: if you want to reverse two cons cells, you can just flip cons so it takes its arguments in the opposite order. However, there's a bit of subtlety at work. Ordinary lists are *right* associative: `(1 2 3 4) = (1 . (2 . (3 . (4 . NIL))))`. If you want to reverse this, you need your fold to be *left* associative: `(reverse '(1 2 3 4)) = (4 . (3 . (2 . (1 . NIL))))`. Try it with a foldr instead of a foldl and see what you get.

There's a whole family of member and assoc functions, all of which can be represented with folds. The particular lambda expression is fairly complicated though, so let's factor it out:

```
(define (mem-helper pred op)
  (lambda (acc next)
    (if (and (not acc)
             (pred (op next)))
        next
        acc)))

(define (memq obj lst)
  (fold (mem-helper (curry eq? obj) id)
        #f
        lst))

(define (memv obj lst)
  (fold (mem-helper (curry eqv? obj) id)
        #f
        lst))

(define (member obj lst)
  (fold (mem-helper (curry equal? obj) id)
        #f
        lst))

(define (assq obj alist)
  (fold (mem-helper (curry eq? obj) car)
        #f
        alist))

(define (assv obj alist)
  (fold (mem-helper (curry eqv? obj) car)
        #f
        alist))

(define (assoc obj alist)
  (fold (mem-helper (curry equal? obj) car)
        #f
        alist))
```

The helper function is parameterized by the predicate to use and the operation to apply to the result if found. Its accumulator represents the first value found so far: it starts out with #f and takes on the first value that satisfies its predicate. We avoid finding subsequent values by testing for a non-#f value and returning the existing accumulator if it's already set. We also provide an operation that will be applied to the next value each time the predicate tests: this lets us customize mem-helper to check the value itself (for member) or only the key of the value (for assoc).

The rest of the functions are just various combinations of eq?/eqv?/equal? and id/car, folded over the list with an initial value of #f.

Next, let's define the functions `map` and `filter`. Map applies a function to every element of a list, returning a new list with the transformed values:

```
(define (map func lst)
  (foldr (lambda (x y)
           (cons (func x) y))
         '()
         lst))
```

Remember that foldr's function takes its arguments in the opposite order as fold, with the current value on the left. Map's lambda applies the function to the current value, then conses it with the rest of the mapped list, represented by the right-hand argument. It's essentially replacing every infix cons constructor with one that conses, but also applies the function to its left-side argument.

`filter` keeps only the elements of a list that satisfy a predicate, dropping all others:

```
(define (filter pred lst)
  (foldr (lambda (x y)
           (if (pred x)
               (cons x y)
               y))
         '()
         lst))
```

This works by testing the current value against the predicate. If it's true, replacing cons with cons, i.e. don't do anything. If it's false, drop the cons and just return the rest of the list. This eliminates all the elements that don't satisfy the predicate, consing up a new list that includes only the ones that do.

We can use the standard library by starting up our Lisp interpreter and typing `(load "stdlib.scm")`:

```
$ ./lisp
Lisp>>> (load "stdlib.scm")
```

```
(lambda ("pred" . lst) ...)
Lisp>>> (map (curry + 2) '(1 2 3 4))
(3 4 5 6)
Lisp>>> (filter even? '(1 2 3 4))
(2 4)
Lisp>>> quit
```

There are many other useful functions that could go into the standard library, including list-tail, list-ref, append, and various string-manipulation functions. Try implementing them as folds. Remember, the key to successful fold-programming is thinking only in terms of what happens on each iteration. Fold captures the pattern of recursion down a list, and recursive problems are best solved by working one step at a time.

12 Conclusion

You now have a working Scheme interpreter that implements a large chunk of the standard, including functions, lambdas, lexical scoping, symbols, strings, integers, list manipulation, and assignment. You can use it interactively, with a REPL, or in batch mode, running script files. You can write libraries of Scheme functions and either include them in programs or load them into the interactive interpreter. With a little text processing via awk or sed, you can format the output of UNIX commands as parenthesized Lisp lists, read them into a Scheme program, and use this interpreter for shell scripting.

There're still a number of features you could add to this interpreter. Hygienic macros let you perform transformations on the source code before it's executed. They're a very convenient feature for adding new language features, and several standard parts of Scheme (such as let-bindings and additional control flow features) are defined in terms of them. Section 4.3 of R5RS defines the macro system's syntax and semantics, and there is a whole collection of papers on implementation. Basically, you'd want to intersperse a function between `readExpr` and `eval` that takes a form and a macro environment, looks for transformer keywords, and then transforms them according to the rules of the pattern language, rewriting variables as necessarily.

Continuations are a way of capturing "the rest of the computation", saving it, and perhaps executing it more than once. Using them, you can implement just about every control flow feature in every major programming language. The easiest way to implement continuations is to transform the program into continuation-passing style, so that eval takes an additional continuation argument and calls it, instead of returning a result. This parameter gets threaded through all recursive calls to eval, but only is only manipulated when evaluating a call to call-with-current-continuation.

Dynamic-wind could be implemented by keeping a stack of functions to execute when leaving the current continuation and storing (inside the continuation data type) a stack of functions to execute when resuming the continuation.

If you're just interested in learning more Haskell, there are a large number of libraries that may help:

- For webapps: WASH, a monadic web framework.
- For databases: HaskellDB, a library that wraps SQL as a set of Haskell functions, giving you all the type-safety of the language when querying the database.
- For GUI programming: Fudgets and wxHaskell. wxHaskell is more of a conventional MVC GUI library, while Fudgets includes a lot of new research about how to represent GUIs in functional programming languages.
- For concurrency: Software Transactional Memory, described in the paper Composable Memory Transactions.
- For networking: GHC's Networking libraries.

This should give you a starting point for further investigations into the language. Happy hacking!