Write You A Scheme, Version 2.0

Adam Wespiser

November 28, 2016

Overview

> *A programming language is for thinking about programs, not for expressing programs you've already thought of. It should be a pencil, not a pen.* **Paul Graham**

Welcome

Welcome to Write You a Scheme, Version 2.0. Github repo . You may be familiar with the original Write Yourself a Scheme in 48 hours by Jonathan Tang, and this is a much needed upgraded version. We use as much modern, industry ready Haskell to implement a Scheme that is well on its way to being ready for production. This series will teach how to create a programming language by walking the reader through the components of a Scheme variant Lisp in Haskell. This should take about a weekend of study/programming for a beginner who might have to look up a few new concepts to really internalize the material. The ideal reader with have some experience in Haskell, and eager to see how all the pieces come together in a medium to large sized project.

We won't have time to go into a lot of detail on things like monad transformers or interpreter theory. Instead, links to further information will be provided when needed. If you are looking for a good Haskell intro, the concepts from Learn You a Haskell for Great Good can be reviewed. Exercism.io has a great collection of Haskell exercises and submission system for getting hands on experience. A great reference while reading this book is What I Wish I Knew When Learning Haskell. Specifically for industry, FP-Complete's Haskell Syllabus is a great guide. The HaskellWiki has a comprehensive list of resources available for learning Haskell.

This a community project, and if you have improvements, please contact me over Github or on Twitter @wespiser. Better yet, submit a PR! We've had to balance language features, tutorial breadth, and level of complexity to provide beginners with a robust but easy to understand implementation. If you have any ideas on how I can better meet these goals, please let us know! We advocate open source languages, although it is possible to use our Scheme to build a vendor-specific language, this article spells out why that is a bad idea. However, if you need an interpreter for commercial purposes, Abstract Definition Interpreters contains the conceptual underpinnings and "invents", or really "discovers", the kind of interpreter that can be used.

Roadmap

Here's the overview of what we will be doing and where we will go:

- Overview What you are reading right now.
- Introduction Scheme syntax and semantics, as well as the Haskell implementation.
- Parser Transformation of text into abstract syntax tree.
- Evaluation Interpretation of abstract syntax tree using monad transformers.

- Errors & Exceptions Exception handling and messages used throughout project. Creation of error messages.
- Primitives Primitive functions that are loaded into the environment.
- REPL Read Eval Print Loop to test out our work so far.
- **IO** Reading and writing to files for both Scheme commands and the reading of program files.
- **Standard Library** Creation of Scheme standard library from primitive functions.
- **Conclusion** We conclude the project, giving some final thoughts on the project.

Why Lisp ?

Although the majority of modern programming follows C/Algo style, Lisp syntax is simple, using the same syntax to represent code and data: the list. This is called homoiconicity, a feature which makes both parsing and evaluation much simpler compared to other languages. Scheme, a dialect of Lisp, is a particularly straight-forward. We won't strictly follow the Scheme standard for the sake of brevity, but aim to include many useful features. If you are interested in what a fully fledged Scheme looks like, Haskell-Scheme is a fully featured language, in Haskell, and Chicken-Scheme implemented in C. Our Scheme contains the basic elements of these more featured languages.

The Scheme Programming Language is a great book to teach the function of an industrial strength Scheme, and reference to explain language features. Another great book to learn Scheme and interpretation theory is Structure and Interpretation of Computer Programs (SICP), which discusses how to build a meta-circular evaluator in Lisp. Lisp has a history as an educational language, and its simplicity is mostly why we will be continuing the tradition here.

While on the topic of Lisp, the Clojure Programming Language is a modern, functional approach to Lisp targeting the JVM. If you are interested in designing your own Lisp, Clojure is a great example of how you can take Lisp to a modern programming environment. It is probably the most supported modern Lisp community, and is a great source of information and ideas. The other industry relevant to Lisp is EmacsLisp, which is a domain specific language for programmers that don't know vim! Finally, Read Eval Print Love contains a wonderful and insightful discussion of the Lisp family, and is a great source of additional information for the interested reader!

Why Haskell ?

First off, why not? I love Haskell! Haskell is a purely functional, typed, compiled, and lazy language with many sophisticated and advanced features. With over two decades of research level and industrial development, Haskell is the focus of numerous computer science research projects and implements these ideas with a production worthy compiler. Who cares? Well, these features give the programmer expressive power above and beyond today's status quo. A set of very powerful and safe abstractions has emerged from its test bed. With great power, comes great responsibility, and there are many complex and under-adopted features that take a while to learn. Fortunately, if you only use a subset of advanced features, Haskell can be an effective production language.

It may be attractive to include advanced type theory extensions and over engineer your project with all the latest advances in type theory. If your background is in academic Haskell research or you have been hacking around in Haskell for years, there is no problem understanding. If you are trying to hire developers to work on your project with you, finding people who can be trained with less cost than the performance gains will be a serious hinderance when proposing your project to weary business type folks. For this reason, we take a conservative approach on what features and abstract concepts are used in this Scheme. When in doubt, we pick the simplest possible abstraction that is needed for functionality, and avoid extra language pragma and type theory extensions at all cost. If you use Haskell for work, your colleague's job will go a lot easier and less technical debt will emerge.

Project Tool Chain

What you need to run the project is in `README.md`, and if you are excited to start, skip to Chapter 1!

Haskell is not an island unto itself, and we must manage the libraries required to build the project. I recommend using Ubuntu, version 14.04 or 16.04, (any Linux distribution should work, contact me if you have problems) and the build tool Stack. The library versions are determined for Stack in scheme.cabal, while stack.yaml is the version of Stack's dependency resolver, and Build.hs is Haskell code for generating documents from these markdown files. More info on stack, here: The README.md contains full instructions on how to build the project source code and documentation, which I encourage you to do. The best way to learn is to modify, break, fix, and finally improve the source code. Two included scripts, build which will monitor for file changes then build upon updates, and run, which will drop you into an interactive REPL, were invaluable in the development of this project. Please feel free to contact with me with any great ideas, modifications, improvements, or vaguely related but interesting concepts. I made this project for you, use it however you please.

What are we going to do?

We are going to make a very basic, but robust, programming language that is both simple to use and highly extendible. You are encouraged to take the language we build and add more functionality. For instance, you could use this language for running jobs on a High Performance Computing Cluster, or analyzing financial data. Language development is really a "killer app" for

Haskell, and the approach we take in this tutorial is the basis you'll need to create a domain specific language for industrial purposes.

Scheme Syntax

Lisp is a list processing language, and is old enough to join the AARP with a very storied history. Scheme is a fully specified Lisp implementation, though we will take many liberties for the sake of simplicity. For Scheme, this means every expression is a list with a prefix operator. This is known as an S-Expression. Whenever a S-Expression is encountered, it is evaluated in the same way, minus a handful of special forms. Data is also represented as an S-Expression, and there is no syntactical difference between code and data. This minimalism is what Scheme is well known for!

Scheme semantics

Similar to Haskell, Scheme is a functional programming language. Everything in our Scheme is an object, for instance numbers, strings, functions, variables, and booleans. This is our Haskell type, `LispVal`. These objects are stored in a single environment, `EnvCtx` which is queried to resolve the value of an evaluated variable. In Scheme, no object is ever destroyed, and we won't need garbage collection. However, the scope of a variable is limited to its lexical context. Further, arguments in scheme are passed by value, so when a function is called, all of its arguments are evaluated before being passed into the function. Contrast this to Haskell, where evaluation is lazy, and values are not computed until they are needed, which is known as "call by need".

The environment to find and store variables is the same to find primitive functions, so when `(file? "tmp")` is evaluated, `file?` is a variable with a corresponding value (a function) in the environment. This approach is a system called Lisp-1, contrasted to Common Lisp's Lisp-2 system, where functions and variables have different environments.

Scheme Type System

Scheme is a dynamic language, like Python, Ruby, Perl, or as contrasted with a static language like C++ or Java. Dynamic languages are easy to use and simple to implement, but allow for some preventable errors that would be impossible in a static language. For an example from our Scheme, `(+ 'a' 1)` is valid syntax wildly open to interpretation at runtime. (Try it on the REPL!)

If you are interested in building a typed language in Haskell, this guide shows how type inference makes language engineering significantly more complex. Dynamic languages are not all doom and gloom: they give the user tremendous flexibility. The R Programming Language is an excellent example of a dynamic language that excels at statistical computing by giving the user incredible flexibility and choice over how to implement ideas.

A concept called Dynamic Dispatch allows functions to be determined, at runtime, by the types of the arguments passed in, so `(+ 1 1)` and `(+ "a" "b")` could use different versions of the `+` function. This is a key feature in dynamically typed programming languages, and we will be implementing this feature in our Scheme.

Interpreted

We are building an interpreted language, an alternative to compiling to assembly language, LLVM or using a virtual machine like Java's JVM. This means that we have a program that actively runs to evaluate a program written in our Scheme. This approach yields slower performance due to the higher memory and processor overhead, but we will be able to finish the project in a single weekend.

For the motivated, Lisp In Small Pieces walks you through over 30 interpreted and 2 compiled versions of Scheme, written in Scheme. You can find the program code here. If you want to write a language with performance in mind, you'll want to use an LLVM backend. I warn you: there be dragons!

On Type System Complexity, Cautionary Tail Type systems are extremely complex to build, and balancing programming productivity versus performance gains is difficult. For instance, Guy Steele has worked on successful languages like Common Lisp and Java, but spent most of the 8 years building Fortress getting the type system right. Steele cited issues with the complexity of the type system when winding down development on Fortress.

Although type systems are complex, it's theoretically possible to create a type system so advanced that programs can have provable properties and abstractions as powerful as those in mathematics. This is far beyond the scope of this tutorial, and the majority of production code written is done in a dynamic language. However, If you're a novice, then this tutorial is a great way to get involved in a very exciting movement that will shape the way of things to come for industry programming. For now, the best we get is an industrial language that, if it compiles, it runs, and this language is Haskell. Finally, Types and Programming Languages by Benjamin C. Pierce is the place to learn the theory and implementation of type systems.

Scheme Examples, Operational Semantics

To get a feel for our Scheme, here is the evaluation of some functions and their arguments. Keep in mind that we must build the abstractions that are capable of evaluating these forms. Both the right and left hand side of the form are represented with `LispVal`.

List Processing

There are three primitive functions for manipulating lists in our Scheme. We will implement them later as part of the standard library and discuss the tradeoffs.
`car` ... `(car '(1 2 3))` => `1`
`cdr` ... `(cdr '(1 2 3))` => `(2 3)`
`cons` ... `(cons 1 '(2 3))` => `(1 2 3)`

Mathematics

Mathematical functions can take 2 or more arguments.
`(* 60 9)` => `540`
`(+ 10 30 2)` => `42`

Quote

`quote` is a special form that delays evaluation on its argument.
`(quote (1 2 3 4))` => `(1 2 3 4)`
`'(1 2 3 4)` => `(1 2 3 4)`

Conditional Statements

The `if` statement acts like it does in any language.
`(if (< 4 5) #f 42)` => `#f`

Lambdas & Anonymous functions

`lambda` is used to create an anonymous function.
`((lambda (y) (+ y 2)) 40)` => `42`

Let Statements

`let` takes two arguments. Its first is a paired list of variables and values. These variables are set to corresponding values, which are then in scope for the evaluation of the second argument.
`(let (x 42) x)` => `42`
`(let (x 2 y 40) (+ x y))` => `42`

Begin

`begin` evaluates a series of one or more S-Expressions in order. S-Expressions can modify the environment using `define`, then subsequent expressions may access the modified environment. Further, when running a Scheme program, its S-Expressions are essentially wrapped in a single begin function. More on this when we go over Eval.hs.
`(begin (define x 413000) (define y (+ x 281)) (+ x y))` => `826281`

The Rest

Although Scheme is a minimal language, this list of functions is not complete. There are two files that contain the rest of the internally defined functions: special forms in Eval.hs, and the primitives in Prim.hs. For a full Scheme specification, see The R5RS Specification. It's not the most modern, but its complete enough to work.

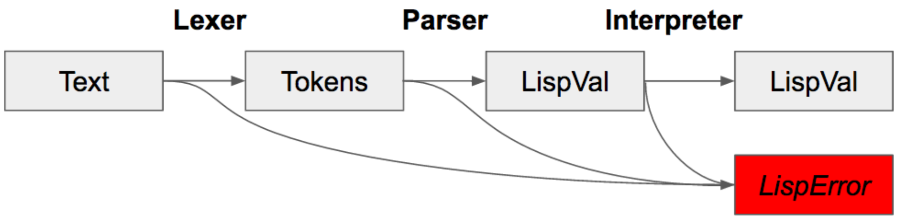
[**Understanding Check**]

- What form does Scheme use to represent data? what about code?
- How would you create a function in Scheme? How about set a variable?
- If Scheme is a Dynamically-Typed Interpreted Functional Language, what does this make C, or your favorite programming language?
- Can you rearrange `let` expressions into `lambda`? What about `lambda` into `let`?
- Write out an explanation and example that demonstrates lexical scope using a `lambda` expression.

Introduction: The Bolts and nuts Of Scheme Interpreters in Haskell

The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language. **Donald Knuth**

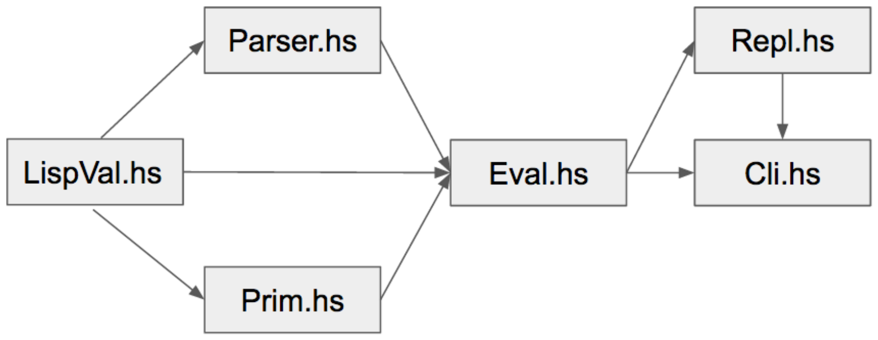
What do we need to build a Scheme?



To make a programming language, we must take user inputed text, turn that text into tokens, parse that into an abstract syntax tree, then evaluate that format into a result. Fortunately, we can use the same structure, `LispVal`, for the abstract syntax tree, which is returned by the parser and also the result of interpretation. Homoiconicity for the win! The lexer and parser are contained in a single library, Parsec, which does most of the work for us. Once we have parsed into `LispVal`, we have to evaluate that `LispVal` into the result of the computation. Evaluation is performed for all valid configurations of S-Expressions, including specials forms like `begin` and `define`. During that computation we need to have an environment for keeping track of bound variables, and an IO monad for reading or writing files during execution. We will also need a way to convert Haskell functions to internal Scheme functions, and a collection of these functions stored in the Scheme environment. Finally, a suitable user interface, including Read/Evaluate/Print/Loop, way to run read files/run programs, and a standard library of functions loaded at runtime defined in Scheme is needed.

This may seem like a lot. But don't worry, all these things, and more, are already available in this project. Together, we'll go through line by line and make sense out of how these Haskell abstractions coalesce to implement a Scheme!

Project Road Map: What do we have?



- **Main.hs** Handles the creation of the binary executable, parsing of command line options.
- **Repl.hs** Read Evaluate Print Loop code.
- **Parser.hs** Lexer and Parser using Parsec code. Responsibility for the creation of LispVal object from Text input.
- **Eval.hs** Contains the evaluation function, `eval`. Patten matches on all configurations of S-Expressions and computes the resultant `LispVal`.
- **LispVal.hs** defines LispVal, evaluation monad, LispException, and report printing functions.
- **Prims.hs** Creates the primitive environment functions, which themselves are contained in a map. These primitive functions are Haskell functions mapped to `LispVals`.
- **Pretty.hs** Pretty Printer, used for formatting error messages. Might drop this, what do you think?

An Engineering Preface

Before we start, there is a note I have to make on efficient memory usage Haskell. The default data structure for Haskell strings, `String`, is quite wasteful in its memory usage. There is an alternative, Data.Text, but to get Haskell to parse strings into `Text` instead of `String` we must use:

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Text as T
```

This declaration will be at the top of every file in the project. Not every library in the Haskell code base has converted to `Text`, so there are two essential helper functions:

```
T.pack :: String -> Text
T.unpack :: Text -> String
```

However, this project uses overloaded strings in all files, and I advocate `Text` becoming the standard for the language. This is the "strong" position on `Text`, and requires that all upstream libraries be written to be either `Text` agnostic, or work with `Text`. This may not always be the case. For these situations, you can convert into `Text` using `T.pack`, or just keep using `String`.

Internal representation, welcome to LispVal

We need a way to represent the structure of a program that can be manipulated with Haskell. Haskell's type system allows for pattern matching on data constructors. Our `eval` function uses this mechanism to differentiate forms of S-Expressions.

LispVal Definition

After much ado, here's the representation of the S-Expression from LispVal.hs. All code and data is represented by one of the following data constructors. There is nothing else. Let's take a look!

```
data LispVal
  = Atom T.Text
  | List [LispVal]
  | Number Integer
  | String T.Text
  | Fun IFunc
  | Lambda IFunc EnvCtx
  | Nil
  | Bool Bool deriving (Typeable)

data IFunc = IFunc { fn :: [LispVal] -> Eval LispVal }
```

`Bool`, `Number` and `String` are straightforward wrappers for Haskell values.
`Nil` is the null type, and the result of evaluating an empty list.
`Atom` represents variables, and when evaluated will return some other value from the environment. To represent an S-Expression we will use `List`, with 0 or more `LispVal`.

Now for the trickier part: functions. There are two basic types of functions we will encounter in Scheme. Primitive functions like `+` use `Fun`. The second type of function is generated in an expression like:

```
((lambda (x) (+ x 100)) 42)
```

To handle lexical scoping, the lambda function must enclose the environment present at the time the function is created. Conceptually, the easiest way is to just bring the environment along with the function. For an implementation, the data constructor `Lambda` accepts `EnvCtx`, which is the lexical environment, as well as `IFunc`, which is a Haskell function. You'll notice it takes its arguments as a list of `LispVal`, then returns an object of type `Eval LispVal`. For more on `Eval`, read the next section. There's also a `deriving (Typeable)`, which is needed for error handling. More on that later!

Evaluation Monad

(from LispVal.hs)

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import qualified Data.Map as Map

import Control.Monad.Except
import Control.Monad.Reader

type EnvCtx = Map.Map T.Text LispVal

newtype Eval a = Eval { unEval :: ReaderT EnvCtx IO a }
  deriving ( Monad
           , Functor
           , Applicative
           , MonadReader EnvCtx
           , MonadIO)
```

For evaluation, we need to handle the context of a couple of things: the environment of variable/value bindings, exception handling, and IO. In Haskell, IO and exception handling are already done with monads. Using monad transformers we can incorporate IO, and Reader (to handle lexical scope) together in a single monad. Using `deriving`, the functions available to each of the constituent monads will be available to the transformed monad without having to define them using `lift`. A great guide about using monad transformers to implement interpreters is Monad Transformer Step by Step. It will start with a simple example and increase complexity.

It's important to keep in mind that evaluation is done for `LispVal`s that are wrapped within the `Eval` monad, which will provide the context of evaluation. The process of `LispVal -> Eval LispVal` is handled by the `eval` function, and this will be discussed a few chapter ahead.

Reader Monad and Lexical Scope We use the `ReaderT` monad to handle lexical scope. ReaderT is basically a monadic action `e -> m a`, which in our case is `EnvCtx -> IO LispVal`. If you are not familiar with monads, or `ReaderT`, you can see the definitions here. `ReaderT` has two basic functions: `ask` and `local`. Within the monadic context, `ask` is a function which gets `EnvCtx`, and `local` is a function which sets `EnvCtx` and evaluates an expression. As you can imagine, we use `ask` to get the `EnvCtx`, `Map.lookup` and `Map.insert` to either get or store variables, then `local` to evaluate our expression with a modified

EnvCtx. If this doesn't make sense yet, that's okay. There is example code on the way!

Show LispVal

While on the topic of LispVal, we can add some code to nicely print out values in LispVal.hs Ideally, we will have functions for both LispVal -> T.Text and T.Text -> LispVal. The latter will be covered in the next section on parsing.

```haskell
instance Show LispVal where
  show = T.unpack . showVal

showVal :: LispVal -> T.Text
showVal val =
  case val of
    (Atom atom)     -> atom
    (String str)    -> T.concat [ "\"" ,str,"\""]
    (Number num)    -> T.pack $ show num
    (Bool True)     -> "#t"
    (Bool False)    -> "#f"
    Nil             -> "Nil"
    (List contents) -> T.concat ["(", T.unwords $  showVal <$>  contents, ")"]
    (Fun _ )        -> "(internal function)"
    (Lambda _ _)    -> "(lambda function)"
```

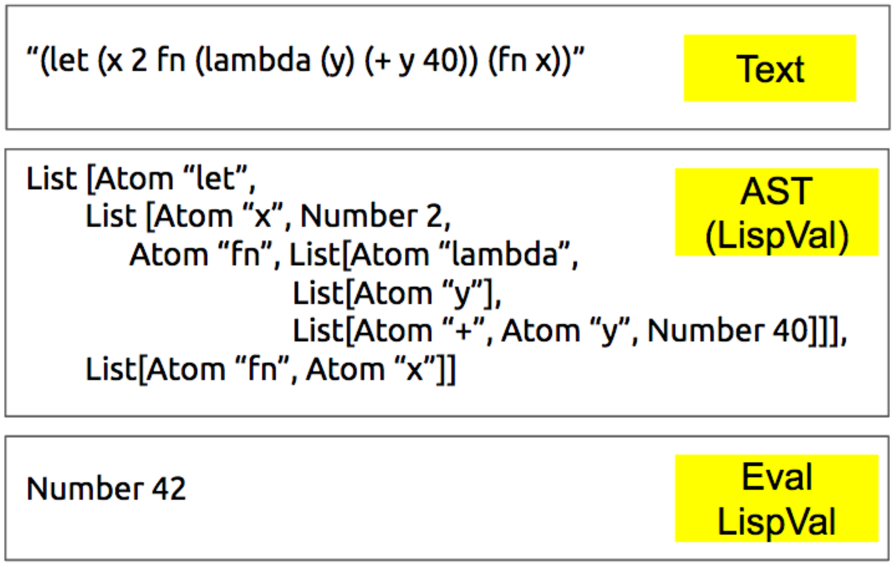
As you can see, we use case to match data constructors instead of pattern matching the arguments of showVal. We have no good way to represent functions as Text, otherwise, LispVal and Text should be interconvertible. This is true before evaluation, or as long as the S-Expression does not contain functions from either data constructor. This feature is analogous to serialization, and later, when we have parsing, we will have de-serialization.

[Understanding Check]

- What's the difference between Text and String?
- How do we represent S-Expressions in Haskell?
- What is a monad transformer? How is this abstraction used to evaluate S-Expressions?
- I mentioned ReaderT gives us e -> m a functionality via runReaderT, imagine if we want e -> m (e, a). What monad would we use, and how would that affect lexical scoping?
- Can you think of some alternative ways to represent S-Expressions? What about GADT? If you would like to see a Haskell language project using GADTs, GLambda is a great project!

Parsing

Writing a really general parser is a major but different undertaking, by far the hardest points being sensitivity to context and resolution of ambiguity. **Graham Nelson**



```
"(let (x 2 fn (lambda (y) (+ y 40)) (fn x))"
```

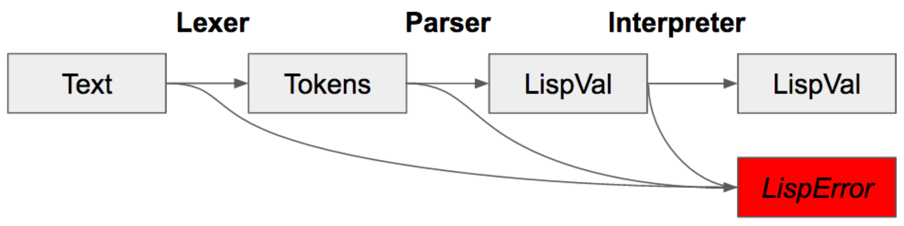
```
List [Atom "let",
      List [Atom "x", Number 2,
            Atom "fn", List[Atom "lambda",
                            List[Atom "y"],
                            List[Atom "+", Atom "y", Number 40]]],
      List[Atom "fn", Atom "x"]]
```

```
Number 42
```

What is Parsing?

First some definitions:

- **lexeme** the basic lexical unit of meaning.
- **token** structure representing a lexeme that explicitly indicates its categorization for the purpose of parsing.
- **lexer** A algorithm for lexical analysis that separates a stream of text into its component lexemes. Defines the rules for individual words, or set of allowed symbols in a programming language.
- **parser** an algorithm for converting the lexemes into valid language grammar. Operates on the level above the lexer, and defines the grammatical rules.



Most basically, Parsing and Lexing is the process of reading the input text of either the REPL or program and converting that into a format that can be evaluated by the interpreter. That converted format, in our case, is `LispVal`. The library we will use for parsing is called `Parsec`.

About Parsec

Parsec is a monadic parser, and matches text to lexeme then parsing that into an abstract syntax via data constructors. Thus, for text input, the lexemes, or units of text that define a language feature, are converted a `LispVal` structure (abstract syntax tree). These lexemes are individually defined via Parsec, and wholly define the valid lexical structure of our Scheme.

Why Parsec?

Parsec is preferable for its simplicity compared to the alternatives: Alex & Happy or Attoparsec. Alex & Happy are more complex, and require a separate compilation step. Parsec works well for most grammars, but is computationally expensive for left recursive grammars. Attoparsec is faster, and preferable for parsing network messages or other binary formats. Parsec has better error messages, a helpful feature for programming languages. If our language required a lot of left-recursive parsing, Alex & Happy would probably be a better choice. However, the simplicity and minimalism of Scheme syntax makes parsing relatively simple. *Note* A new library has been created from a fork of Parsec called MegaParsec, it takes a monad transformer approach to parsing, and appears quite feature rich and claims to be industry ready!

How parsing will work

The parser will consume text, and return a `LispVal` representing the abstract syntax tree that can be evaluated into an `Eval LispVal`. Parsec defines parsers using,

```haskell
newtype Parser LispVal = Parser (Text -> [(LispVal,Text)])
```

Thus, a Parser is a type consisting of a function that 1) takes some `Text` and 2) return a `LispVal` and some `Text`

Parser Imports

Parser.hs

```haskell
import LispVal
import Text.Parsec
import Text.Parsec.Text
import Text.Parsec.Expr
import qualified Text.Parsec.Token as Tok
import qualified Text.Parsec.Language as Lang
import qualified Data.Text as T
import Control.Applicative hiding ((<|>))
import Data.Functor.Identity (Identity)
```

Good for us, Parsec is available with `Text`, not just `String`. If you are looking online for examples of Parsec, make sure you have the correct encoding of strings. Converting can be a little bit of a hassle, but its worth the extra effort. If you are really serious about your language project, I suggest using Alex & Happy. I've happily used them in production to parse a javascipt-esque language!

Lexer

```haskell
lexer :: Tok.GenTokenParser T.Text () Identity
lexer = Tok.makeTokenParser style

style :: Tok.GenLanguageDef T.Text () Identity
style = Lang.emptyDef {
  Tok.commentStart = "{-"
  , Tok.commentEnd = "-}"
  , Tok.commentLine = "--"
  , Tok.opStart = Tok.opLetter style
  , Tok.opLetter = oneOf ":!#$%%&*+./<=>?@\\^|-~"
  , Tok.identStart = letter <|>  oneOf "-+/*=|&><"
  , Tok.identLetter = digit <|> letter <|> oneOf "?+=|&-/"
  , Tok.reservedOpNames = [ "'", "\""]
  }
```

Whelp, that's about all we need. Parsec does the heavy lifting for us, all we need to do is supply the specification for the lexeme. Starting with comments, we'll use the same standards as Haskell, and moving on to operators and identifiers. Finally, we established reserved operators, which will be single and double quotes.

```haskell
-- pattern binding using record destructing !
Tok.TokenParser { Tok.parens = m_parens
           , Tok.identifier = m_identifier } = Tok.makeTokenParser style
```

Before we move on, I'm going to use record deconstruction and pattern binding to define some shortcuts. It's a neat trick!

Parser

```haskell
reservedOp :: T.Text -> Parser ()
reservedOp op = Tok.reservedOp lexer $ T.unpack op
```

Using our shortcut, we can define a quick helper function to lex the input text for reserved operators.
Now, given the diagram, we must move from `Text` to `LispVal`. Parsec will be handling the lexer algorithm, which leaves the formation of `LispVal`. You will notice the use of `T.pack`, which is `T.pack :: String -> Text`. For each data constructor of type `LispVal`, we will have a separate parsing function. Later, we will combine them into a single parser for all S-Expressions called `parseExpr`.

```haskell
parseAtom :: Parser LispVal
parseAtom = do
  p <- m_identifier
  return $ Atom $ T.pack p

parseText :: Parser LispVal
parseText = do
  reservedOp "\""
  p <- many1 $ noneOf "\""
  reservedOp "\""
  return $ String . T.pack $  p

parseNumber :: Parser LispVal
parseNumber = Number . read <$> many1 digit

parseNegNum :: Parser LispVal
parseNegNum = do
  char '-'
  d <- many1 digit
  return $ Number . negate . read $ d

parseList :: Parser LispVal
parseList = List . concat <$> Text.Parsec.many parseExpr
                                `sepBy` (char ' ' <|> char '\n')

parseSExp = List . concat <$> m_parens (Text.Parsec.many parseExpr
                                       `sepBy` (char ' ' <|> char '\n'))

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

```haskell
parseReserved :: Parser LispVal
parseReserved = do
  reservedOp "Nil" >> return Nil
  <|> (reservedOp "#t" >> return (Bool True))
  <|> (reservedOp "#f" >> return (Bool False))
```

Phew! That wasn't so bad! Monadic parsing makes this somewhat manageable. We consume a little bit of text, grab what we need with monadic binding, maybe consume some more text, then return our `LispVal` data constructor with the bound value. There is one extra parser, `parseList`. This is used to parse programs since programs can be a list of newline delimited S-Expressions. Now that we can parse each of the individual `LispVals`, how would we parse an entire S-Expression with `parseExpr`?

```haskell
parseExpr :: Parser LispVal
parseExpr = parseReserved <|> parseNumber
  <|> try parseNegNum
  <|> parseAtom
  <|> parseText
  <|> parseQuote
  <|> parseSExp
```

Of course! `<|>` is a combinator that will go with the first parser that can successfully parse into a `LispVal`. If you are writing this parser, or don't like mine and decide to write your own (do it!), this part will require some thought. Here be dragons, and if your syntax is very complex, Alex & Happy provide a nice alternative way to define syntax.

[Understanding Check]

You just moved to Paris, France and can't find the "quote" on a local keyboard. Change the "quote" to a "less than" symbol in the parser. (single or double, I can't find either one on european keyboards) Where do all the changes need to be made?
Move around the ordering in `parseExpr`, what happens? When do things break?
`parseNumber` works for positive numbers, can you get it to work for negative numbers?
Parsec `Parser` is used as both a monad and a functor, give an example of both.

Putting it all together

Parsec needs to play nice with the rest of the project, so we need a way to run the parser on either text from the REPL or a program file and return `LispVal` or `ParseError`. There's a monad for that!

```haskell
contents :: Parser a -> Parser a
contents p = do
  Tok.whiteSpace lexer
  r <- p
  eof
  return r

readExpr :: T.Text -> Either ParseError LispVal
readExpr = parse (contents parseExpr) "<stdin>"

readExprFile :: T.Text -> Either ParseError LispVal
readExprFile = parse (contents parseList) "<file>"
```

`contents` is a wrapper for a Parser that allows leading whitespace and a terminal end of file (eof). For `readExpr` and `readExprFile` we are using Parsec's `parse` function, which takes a parser, and a `Text` input describing the input source. `readExpr` is used for the REPL, and `readExprFile`, which uses our `parseList` and can handle newline or whitespace delimmited S-Expressions, for program files.

Conclusion From the top, we have gone from text input, to tokens, to `LispVal`. Now that we have `LispVal` representing the abstract syntax tree, we need to get to `Eval LispVal`, the final computed value. If you'd like to see the parser in action, run `stack test`, where the parsing tests are the first block in test-hs/Spec.hs. Now, it's time to start running programs, let's take it to eval and see how `LispVal` gets computed!

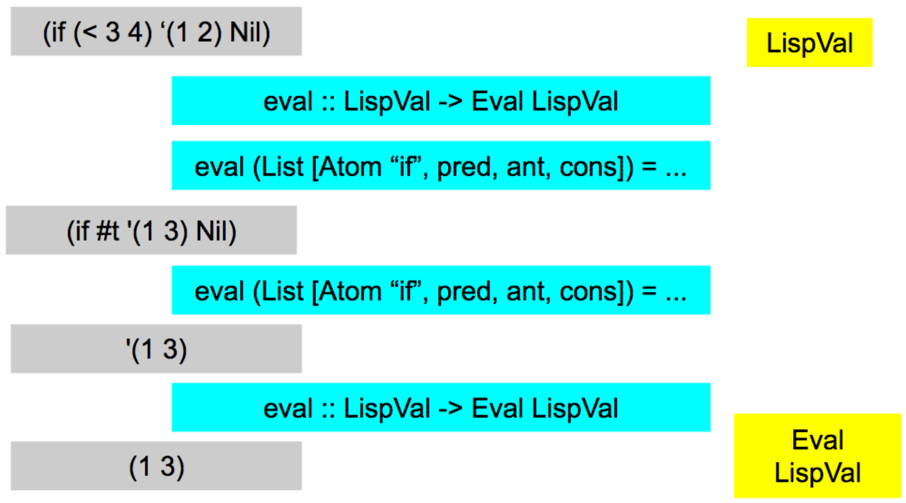
Next, Evaluation home...back...next

Additional Reading Monadic parsing is a little tricky to understand. In terms of the rest of programming languages, it's a somewhat orthogonal subject but nonetheless absolutely necessary to build a programming language!

- https://github.com/bobatkey/parser-combinators-intro
- http://unbui.lt/#!/post/haskell-parsec-basics

Evaluation

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information. **Winston Churchill**


(if (< 3 4) '(1 2) Nil)
LispVal
eval :: LispVal -> Eval LispVal
eval (List [Atom "if", pred, ant, cons]) = ...
(if #t '(1 3) Nil)
eval (List [Atom "if", pred, ant, cons]) = ...
'(1 3)
eval :: LispVal -> Eval LispVal
(1 3)
Eval LispVal

Evaluation Context

LispVal.hs defines our key data structure for evaluation:

Eval Monad

```haskell
newtype Eval a = Eval { unEval :: ReaderT EnvCtx IO a }
  deriving (Monad
           , Functor
           , Applicative
           , MonadReader EnvCtx
           , MonadIO)
```

This code defines the data structure used to control evaluation. Specifically, it allows for lexical scoping, user input, and will contain the return `LispVal` for any valid expression. If your asking yourself how such a complex data structure is possible, then welcome to the world of monad transformers. These wonderful abstractions allow programmers to combine monads and automatically generate the correctly lifted functions. For instance, our `Eval`'s `IO` would need its functions to be lifted, but since we derived `MonadIO`, this happens automatically during compilation.
This 'magic' is made possible through `{-# LANGUAGE GeneralizedNewtypeDeriving #-}` pragma. The big addition we get here is `liftIO`, which lets us operate `IO`

actions within `Eval` context: `liftIO :: IO a -> m a`. We will be discussing this more in Chapter 7 on input and output.

Running The Eval Monad

Getting the `EnvCtx`

```
basicEnv :: Map.Map T.Text LispVal
basicEnv = Map.fromList $ primEnv
        <> [("read" , Fun $ IFunc $ unop $ readFn)]
```

This is our primitive environment, which will be detailed in the next chapter on Prim.hs. Recall that we defined `EnvCtx` in LispVal.hs as:

```
type EnvCtx = Map.Map T.Text LispVal
```

Our environment is a collection of bindings between names and entities, referenced name. For now, we must only concern ourselves with the fact that the names in the environment comes from `LispVal's Atom` data constructor. This data structure is going to be the basis of our lexically scoped variable look up.

```
evalFile :: T.Text -> IO () --program file
evalFile fileExpr = (runASTinEnv basicEnv $ fileToEvalForm fileExpr)
                     >>= print

fileToEvalForm :: T.Text -> Eval LispVal
fileToEvalForm input = either (throw . PError . show )
                              evalBody
                              $ readExprFile input

runParseTest :: T.Text -> T.Text -- for view AST
runParseTest input = either (T.pack . show)
                            (T.pack . show)
                            $ readExpr input

runASTinEnv :: EnvCtx -> Eval b -> IO b
runASTinEnv code action = runResourceT
                          $ runReaderT (unEval action) code
```

From Parser.hs we have

```
readExprFile :: T.Text -> Either ParseError LispVal
readExprFile = parse (contents parseList) "<file>"
```

There is a lot of movement here (possibly dragons), and the functions above do the following things:

- `evalFile` is used from the `main :: IO ()` loop to run a program file.

- `readExprFile` runs the parser on the program text to return a `LispVal` or `ParseError`.
- `fileToEvalForm` runs the parser, if an error occurs, it converts it into a `LispException`, `PError` and throws it, else, it evaluates it using `evalBody`.
- `runASTinEnv` executes the the `evalBody :: LispVal -> Eval Body` with the `EnvCtx`, essentially running our program by first unwrapping `Eval` with `unEval` (the data accessor to `Eval`), then using the `runReaderT` and `runResourceT` functions on the transformed monad.

eval function: rationale

Using our aptly named `Eval` structure, we will define the `eval` function within Eval.hs as follows:

```
eval :: LispVal -> Eval LispVal
```

Given our type signature, we can think back to our Scheme semantics, and recall we need to handle a few special cases:

- **LispVal** For primitive types, like Number or String, these values will evaluate to themselves. This is known as the autoquote facility of Lisp.
- **begin** The begin function takes a series (one or more) S-Expressions, starting with 0 or more `define` statements, and evaluates each form in order. The argument to `begin` will be known ad a "body" or "body-expression".
- **define** Binds an evaluated `LispVal` to an `Atom`.
- **write** Takes its argument, un-evaluated, and returns a `String` whose value is the result of running `showVal`.
- **if** Evaluates its first form, if true, evaluates second, else, evaluates the third. The same as every other language!
- **let** Let takes a list of alternating atoms and S-Expressions and another S-Expression, which is a body expression. The first argument of alternating values and S-Expressions is bound to a local environment which the body can access when evaluated. No order of evaluation on the first argument can be assumed, thus variables to be bound cannot appear in the S-Expression bound to another variable.
- **lambda** This is how we will create anonymous functions. Thinking back to our `LispVal`, recall the data constructor, `Lambda IFunc EnvCtx` where essentially `IFunc :: LispVal -> Eval LispVal`. Two arguments are accepted, a list of atomic values to serve as the function parameters, and a body-expression, which can be evaluated in an environment with the incoming arguments bound to the parameters from the first argument.
  The `EnvCtx`, and thus the lexical scope, which can store the local environment via the `reader` monad function, `ask`. Lambda is really powerful, so incredibly powerful, that we could use it with `Atom` to implement everything else. It's a neat idea called Lambda Calculus!

These will be known as our "Special Forms", different from functions defined in the primitive environment or standard library, as they are implemented via pattern matching on the argument (`LispVal`) of the `eval` function. All other Scheme functions will either be primitives or from the standard library, and have their argument evaluated before being passed into the function. Therefore, special forms will require non-standard evaluation of their arguments, sometimes having arguments not evaluated, or evaluated under special conditions. This difference is what determines what must be implemented as an `eval` pattern match, and what can be done elsewhere.

Eval Function: implementation

```
eval :: LispVal -> Eval LispVal
```

The eval function is the heart of our interpreter, and must be able to pattern match every possible valid syntax, as well as the special forms. This is a pretty tall order, so we are going to approach this by going through the eval function piece by piece along with the helper functions needed to run that code. It's a little disjointed, but the simplest way to explain exactly how to implement the syntax and semantics of Scheme in Haskell. As always, to view it all together, see Eval.hs. As we go through the code you will see some `throw`, which are covered in the next chapter, for now, recognize that `throw $ LispExceptionConstructor "message-1"` returns `Eval LispVal`.

quote

```
eval (List [Atom "quote", val]) = return val
```

Quote returns an un-evaluated value, pretty basic, but it helps to start out simple!

autoquote

```
eval (Number i) = return $ Number i
eval (String s) = return $ String s
eval (Bool b)   = return $ Bool b
eval (List [])  = return Nil
eval Nil        = return Nil
```

For Number, String, Bool, and Nil, when we evaluate, we simply return the value. This is known as the autoquote facility, and makes it so we can pass these values into functions and evaluate them without consequence, or logic deferring evaluation of these types. For `List`, we have made the evaluation of an empty list go to `Nil`. This will be useful for functions that consume the input of a list conditional on the list having items left.

write

```
eval (List [Atom "write", rest]) =
          return . String . T.pack $ show rest

eval (List ((:) (Atom "write") rest)) =
          return . String . T.pack . show $ List rest
```

Write does not evaluate argument or arguments, and instead runs `show` on them before return that value in a `String`. For the second version, we are taking the two or more arguments passed to write and simply converting them into a `List`.

Atom

```
eval n@(Atom _) = getVar n

getVar :: LispVal ->  Eval LispVal
getVar (Atom atom) = do
  env <- ask
  case Map.lookup atom env of
      Just x  -> return x
      Nothing -> throw $ UnboundVar atom
```

Now we're talking! When we evaluate an Atom, we are doing variable lookup. `getVar` will do this lookup by getting the `EnvCtx` via `ask`, then running a `Map` lookup, returning the value if found, else throwing an exception.

if

```
eval (List [Atom "if", pred, truExpr, flsExpr]) = do
  ifRes <- eval pred
  case ifRes of
      (Bool True)  -> eval truExpr
      (Bool False) -> eval flsExpr
      _            -> throw $ BadSpecialForm "if"
```

Here we implement the familiar `if` special form. First, we evaluate the predicate by recursing on the `eval` function. Given that value, we pass it to a case statement, for True, then we evaluate the third S-Expression, and for false, the fourth. Thus, `if` only evaluates one of its third or fourth arguments, requiring it to be a special form.

let

```
eval (List [Atom "let", List pairs, expr]) = do
  env   <- ask
  atoms <- mapM ensureAtom $ getEven pairs
  vals  <- mapM eval       $ getOdd  pairs
  let env' = Map.fromList (Prelude.zipWith (\a b -> (extractVar a, b)) atoms vals) <> env
  in local (const env')  $ evalBody expr
```

```
getEven :: [t] -> [t]
getEven [] = []
getEven (x:xs) = x : getOdd xs

getOdd :: [t] -> [t]
getOdd [] = []
getOdd (x:xs) = getEven xs

ensureAtom :: LispVal -> Eval LispVal
ensureAtom n@(Atom _) = return  n
ensureAtom n = throw $ TypeMismatch "atom" n

extractVar :: LispVal -> T.Text
extractVar (Atom atom) = atom
```

The `let` special form takes two args, a list of pairs and an expression. The list of pairs consists of an atom in the odd place, and an S-Expression in the even place. For instance, `(let (x 1 y 2)(+ x y))` . From the current environment, a new environment is created with these bindings added, and the expression is evaluation as a body-expression in that environment. This is possible with the `local` function, defined along with our monad transformer stack.

begin, define & evalBody

```
eval (List [Atom "begin", rest]) = evalBody rest
eval (List ((:) (Atom "begin") rest )) = evalBody $ List rest

eval (List [Atom "define", varExpr, expr]) = do
  varAtom <- ensureAtom varExpr
  evalVal <- eval expr
  env     <- ask
  let envFn = const $ Map.insert (extractVar varAtom) evalVal env
  in local envFn $ return varExpr

evalBody :: LispVal -> Eval LispVal
evalBody (List [List ((:) (Atom "define") [Atom var, defExpr]), rest]) = do
  evalVal <- eval defExpr
  env     <- ask
  local (const $ Map.insert var evalVal env) $ eval rest
evalBody (List ((:) (List ((:) (Atom "define") [Atom var, defExpr])) rest)) = do
  evalVal <- eval defExpr
  env     <- ask
  let envFn = const $ Map.insert var evalVal env
  in local envFn $ evalBody $ List rest
evalBody x = eval x
```

The begin special form is designed to accept two types of inputs: one for each of the pattern matched `evals`. In the first form, `eval` matches on `List[Atom "begin", rest]`, where rest is then passed to `evalBody`. The second form, or expanded form, accepts `List ((:) (Atom "begin") rest)`, where rest is type `[LispVal]`. `rest` is then moved into a `LispVal` via the `List` data constructor, and then passed onto `evalBody`. This allows us to either accept the arguments as a single list, or as individual arguments. Further, this prevents us from wrapping an extra `List` around single values, since we make sure there are at least two or more values in `rest` when we pass it to `List`, before subsequently passing to `evalBody`. There is one more pattern match, for define, but its real worth will only emerge after discussing `evalBody`.

`evalBody` allows us to use the `define` statement, and evaluate body-expressions. These body expressions are not only important to `lambda` and `let`, but absolutely essentially to the creation of a standard library. The standard library consists of Scheme files with many defines used to bind library functions to the environment. `evalBody` pattern matches on a list composed of either (a define expression, and one other expressions) or (a define expression, and two or more other expressions). For the first definition of `evalBody`, we pattern match the `define`, insert the matched variable and value to the environment, then run the second expression, `rest`, in this modified environment using `eval` all within a `local` function. This is our base case.
In the case of the second expression, `rest` is not a single expression, but a list of expressions. We handle this by doing the same pattern match and environment update on define, but instead of calling `eval` we wrap `rest` in a `List` and recurse on `evalBody`. Thus, we can subsequently consume define statements.

lambda & applyLambda

```haskell
eval (List [Atom "lambda", List params, expr]) = do
  envLocal <- ask
  return  $ Lambda (IFunc $ applyLambda expr params) envLocal
eval (List (Atom "lambda":_) ) = throw $ BadSpecialForm "lambda"

applyLambda :: LispVal -> [LispVal] -> [LispVal] -> Eval LispVal
applyLambda expr params args = do
  env <- ask
  argEval <- mapM eval args
  let env' = Map.fromList (Prelude.zipWith (\a b -> (extractVar a,b)) params argEval) <> env
  in local (const env' ) $ eval expr

-- From LispVal.hs
data IFunc = IFunc { fn :: [LispVal] -> Eval LispVal }
```

The lambda special form gives our users the ability to create and define functions. Variables within those functions will be lexically scoped, which means they will always have the value they had when they were enclosed within the lambda. We will achieve lexically scoped variables by bringing a copy of the `EnvCtx` with us into the `Lambda` data constructor, `Lambda IFunc EnvCtx`. When we encounter the lambda special form in eval, we grab the environment, then return the data constructor for `Lambda` along an `IFunc $ applyLambda expr params` and the grabbed environment. Looking at `applyLambda`, what we are doing is partially applying the parameters of a function, to return a function that accepts arguments `[LispVal]`, binds them to the atomic values in the parameters, then evaluates the expression passed into `applyLambda` as `expr`. Thus, we go from `applyLambda :: LispVal -> [LispVal] -> [LispVal] -> Eval LispVal` to `applyLambda expr params :: [LispVal] -> Eval LispVal` which is the correct type for the type `IFunc`. Thus, we can achieve lexical scoping using the `ReaderT` monad.

application

```
eval (List ((:) x xs)) = do
  funVar <- eval x
  xVal   <- mapM eval  xs
  case funVar of
      (Fun (IFunc internalFn)) -> internalFn xVal
      (Lambda (IFunc internalfn) boundenv) -> local (const boundenv)
                                                   $ internalfn xVal
      _                -> throw $ NotFunction funVar
```

The final form is application, we've made it! If you are familiar with lambda calculus, this is one of the three forms, along with variables and lambdas. The way we perform application is to pattern match on the head, then tail of `List`. Next, we evaluate both of these values. We run `case` on `funVar`, the head of the list, which should be either a `Fun` (internal function), or `Lambda`, a user-defined or library function. If internal, we simply extract the function of type `[LispVal] -> Eval LispVal` and apply the arguments. For `Lambda`, we must evaluate the function within the environment provided by the `Lambda` to ensure lexical scope is maintained.

Conclusion

That was a lot! If `LispVal` defines the syntax, then `Eval` defines the semantics. Now is a good time to read Eval.hs and see it work all together, since that's all that defines the mechanism of evaluation. Our implementation is made possible by monad transformers, specifically the integration of `ReaderT`, which we use to implement lexical scope. Without monads, we would have to pass in an extra argument to every `eval`, as well as some of the helper functions, not to mention the complication of handling the other functionality of monads composed within our `Eval`. For a simple interpreter, its hard to do much better. For a faster interpreter, we would need to compile.
Anyway, we have the the basis for our language and could start work on proving theoretical properties. We won't do this, and instead move towards gaining the

next thing needed to be practical, a collection of basic operations to manipulate data. But before moving on to primitives, we quickly cover errors, which show up all over `eval` function. If you just can't wait to define some functions and get the REPL up and running, the basic message is, when bad things happen, throw an error that gives the user enough information to fix the problem.

[Understanding Check]

Implement a delay function as a special form that returns its argument as the body of a lambda expression that accepts no arguments. `(delay x) => (lambda () x)`
You careful read the R5RS standard and are upset to realize we have implemented `let` incorrectly. Instead of `(let (x 1 y 2) (+ x y))` the standard calls for `(let ((x 1) (y 2)) (+ x y))`. Make this change.
GADTs are sometimes used to implement `LispVal`. How would evaluation change?
Implement one of (`case`, `letrec`, `let*`, `sequence`, `do`, `loop`) as a special form. Check the R5RS standard for more information.
[Bonus] Find a unique difference in the implementation of special forms between this project and R5RS, implement the special form, then submit a new PR.

Danger Will Robinson, on to Errors! home...back...next

Additional Resources:

- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.596&rep=rep1&type=pdf
- https://github.com/justinethier/husk-scheme/tree/master/hs-src/Language/Scheme

Alternative Implementations

- Extensible Effects. An Alternative to Monad Transformers

Error Checking and Exceptions

Illusions of control are common even in purely chance situations. They are particularly likely to occur in setting that are characterized by personal involvement, familiarity, foreknowledge of the desired outcome, and a focus on success. **Suzanne C. Thompson**

Error Exceptions and All That!

When the user enters incorrect input, we can say they have made an error, and people understand this statement as a general term. However, we are in the business of engineering, and there is a technical distinction between errors and exceptions. Errors, refer to situations where our project cannot handle itself, and we must change the source code to remedy the situation. These are unexpected situations. Exceptions, on the other hand, represent expected, but still irregular situations that we can control. Exceptions can represent problems with a potential Scheme program, like an error parsing, or a bad special form. Haskell Wiki.
The sources for errors and exceptions in Haskell are as follows:
Exceptions: `Prelude.catch`, `Control.Exception.catch`, `Control.Exception.try`, `IOError`, `Control.Monad.Error`
Errors: `error`, `assert`, `Control.Exception.catch`, `Debug.Trace.trace`.
`error` is syntactical sugar for undefined.
Try to keep these in mind, but expect to see only exceptions, unless I've made an error in my assertion, in which case I'll have to trace through and catch my mistake.
We say an exception is checked when it after it is "thrown", another part of code "handles" or "catches" it. This is how our exception system in Haskell will work.

Exceptions Everywhere!

Undefined, unexpected, and generally out of control situations are present in any kind of large system, especially one interacting with the outside world or dealing with use input as complex and complicated as a programming language. Control is an illusion. For our system, we must accept user input, determine that it is valid Scheme syntax, then compute that abstract syntax into a final value. During this process we may interact with the file system or network. It is especially important for programming languages to report and describe the nature of the irregularity.
Thus, there are three types of exceptions that exist in our implementations of Scheme: **Parsing, Evaluation, and IO**. Each of these originate in a distinct type of activity the parser or interpreter is undergoing, but all of them are end up going through the `Eval` monad and are caught and displayed in the same place. (see Eval.hs)

```
someFun :: GoodType -> Eval LispVal
someFun (BadMatch x) = return $ throw $ LispExceptionConstructor "message we send"

someOtherFun x = if (predicate on x) goodThing else throw $ badThingException
```

```
safeExec :: IO a -> IO (Either String a)
safeExec m = do
  result <- Control.Exception.try m
  case result of
    Left (eTop :: SomeException) ->
      case fromException eTop of
        Just (enclosed :: LispException) -> return $ Left (show enclosed)
        Nothing                -> return $ Left (show eTop)
    Right val -> return $ Right val
```

Above we have the code to catch an exception. We use `Control.Exception.try`, then subsequently `fromException` and `case` to take an exception and unwrap it into a `LispException`. For running programs, we might not need to catch an exception thrown in the code, displaying the exception is enough for the user to fix the problem. However, for the REPL, it would be a major pain if we required our users to restart the REPL every time they made a mistake while prototyping a new idea.

Defining an Exception

For our Scheme, an exception will be defined for a internal misuse of a function, a user deviation from accepted syntax or semantics, or the request of an unavailable external resource. Exceptions are thrown in the monad transformer stack, `Eval`, and caught with the `safeExec` function, which is convenient for us, because we can throw exceptions from any function return `Eval LispVal`, which is most of our evaluation code!

```
data LispException
  = NumArgs Integer [LispVal]
  | LengthOfList T.Text Int
  | ExpectedList T.Text
  | TypeMismatch T.Text LispVal
  | BadSpecialForm T.Text
  | NotFunction LispVal
  | UnboundVar T.Text
  | Default LispVal
  | PError String -- from show anyway
  | IOError T.Text
```

Each of these data constructors distinguish the source of their error. Whenever a `LispException` is created, it is then immediately thrown. Ideally, they provide useful information to the user on how to debug their program after a LispException is returned. Though not as descriptive as a fully featured language, we do have the capacity to provide a fair amount of information, including the a custom message and LispVal that are not compatible. The key to improvement here is keeping track and passing more information to LispException when it is created then thrown.

```
instance Show LispException where
  show = T.unpack . showError

unwordsList :: [LispVal] -> T.Text
unwordsList list = T.unwords $  showVal <$> list

showError :: LispException -> T.Text
showError err =
  case err of
    (IOError txt)            -> T.concat ["Error reading file: ", txt]
    (NumArgs int args)       -> T.concat ["Error Number Arguments, expected ", T.pack $ show i
    (LengthOfList txt int) -> T.concat ["Error Length of List in ", txt, " length: ", T.pack
    (ExpectedList txt)       -> T.concat ["Error Expected List in funciton ", txt]
    (TypeMismatch txt val) -> T.concat ["Error Type Mismatch: ", txt, showVal val]
    (BadSpecialForm txt)     -> T.concat ["Error Bad Special Form: ", txt]
    (NotFunction val)        -> T.concat ["Error Not a Function: ", showVal val]
    (UnboundVar txt)         -> T.concat ["Error Unbound Variable: ", txt]
    (PError str)             -> T.concat ["Parser Error, expression cannot evaluate: ",T.pack
    (Default val)            -> T.concat ["Error, Danger Will Robinson! Evaluation could not p
```

Similar to our showVal, from Chapter 1, we override the show Typeclass to give a custom message. The showError has a special case for PError, which uses String and just wraps the error message from the parser. The next source IO, can also be tricky. Although we have the ability to throw an IOError, if there is an unchecked exception during IO operations, it will fall through and not be handling via our LispException pathway.

Conclusion

Accidents will happen, and so we have exception handling for IO, parsing, and evaluation exceptions via our LispException type and our handy Eval monad transformer stack. Exceptions are realized everywhere we have functions that evaluate Scheme code, so handling them is composed into our Eval monad. Verbose exception messages are vital to usability, and we must report enough information to pinpoint the user's misuse of proper syntax, semantics, or resource request. Our main liability with the current monad transformer stack, is that IO can throw an error that is not caught, an unchecked exception. However, we are only using IO to read files, not maintaining open connections for long periods of time, or dispatching concurrent operations on shared resources, so our exposure is minimal. If this is a major concern for you, read the section on "Alternative Exceptions", which discusses other ways to handle exceptions in Haskell.

[Understanding Check]

Go through Eval.hs, find an `LispException` used in a few places and replace it with a new error that is more specific, or merge two `LispException` that are better served by a single constructor. Include support for `showError`.
Many programming languages have information like, "line 4, column 10: variable x is not bound". How would you go about adding line and column information to messages passed to `throwError` in Eval.hs?
We are taking a basic approach to format error messages: `T.concat` and `show`. Text.PrettyPrint offers a rich way to display messages using pretty print combinators. Implement a PrettyPrint interface for `LispException` that provides a uniform interface.

Alternative Exceptions (skippable)

We are handling errors in a very basic way. The use of `IO` causes some trickiness that we won't be able to handle. Here's why:

- **async exceptions** We entirely avoid this topic, but they cannot be ignored for most industrial applications. Control.Concurrent.Async defines the library, which makes it a little bit more difficult to run a few threads, then ignore an exception thrown in a thread while other threads exit successfully. Exceptions in asynchronous threads are known as **asynchronous exceptions**.
- **ExceptT someErrorType IO a** considered bad Exceptions Best Practices. The authors list three reasons why this is considered an anti pattern: 1) Its noncomposable, we see this when we have to do add in the Parser error, and the information in the parser error is not completely congruent with the information we pass to other error messages. 2) It gives the implication that only `LispException` can be thrown. This is true, during `slurp` or any `IO` operation, an error can be thrown that will not be caught. 3) We haven't limited the possible exceptions, we've just added `throwError` or `liftIO . throwIO`.
- **enclosed exceptions**](https://github.com/jcristovao/enclosed-exceptions) FP complete's Catching All Exceptions. . The goal is to catch all exceptions that arise from code.
- a plethora of options for error/exception handling in Haskell:
  1. Control-Monad-Trans-Except
  2. Control-Monad-Error
  3. Control-Monad-Catch
  4. Control-Monad-Except
  5. Control-Exception
  6. UnexpectionIO

The list is pretty daunting, and building upon the approach taken here is a good path forward. If conceptual simplicity is highly valued, biting the bullet and just using the anti-pattern `ExceptT (LispException IO) a` might not hurt too bad. Its not super great code, but its pretty simple to understand, and forces the `either error value` logic to happen during monadic evaluation. `IO` really seems to be tricky here, and if we were running multiple threads, we ought to be handling `async` exceptions. There's no right answer, but a consensus has formed around the opinions exposed in Exceptions Best Practices.

Next, Let's make some functions!

The Primitive Environment

Primitive Strategy

The primitive environment is defined in Prim.hs
Our basic strategy is to create a list of tuples, `[(T.Text,LispVal)]`, where the text is the name of the primitive, and the `LispVal` is a `Fun` representing the internal function. We can use `Map.fromList` to convert this to a `Map` which is our environment for evaluation. To create our `Fun`, we must map an internal Haskell function of some type to a `[LispVal] -> Eval LispVal`. In the process, we must pattern match to get the corresponding `LispVal`s of the correct types, extract the values, and apply our Haskell function. To help with this, we have created `binop`, `binopFold`, and `unop`. This process is complicated to talk our way through, so I will go through an example in the subsequent sections to show how the type signatures reduce.

Full Definition

First, we'll take a look at what everything looks like all together. What we see is that a Haskell function is wrapped with a function, like `numOp` or `numCmp`, which pattern matches on `LispVal` to get the internal Haskell values. This is then wrapped again by `binopFold` or `unop`, which convert the type of the argument to `[LispVal] -> Eval Lisp`, regardless of the number of arguments in the function `numCmp` or `numOp`.

```haskell
mkF :: ([LispVal] -> Eval LispVal) -> LispVal
mkF = Fun . IFunc

primEnv :: Prim
primEnv = [   ("+"     , mkF $ binopFold (numOp    (+))  (Number 0) )
            , ("*"     , mkF $ binopFold (numOp    (*))  (Number 1) )
            , ("++"    , mkF $ binopFold (strOp    (<>)) (String ""))
            , ("-"     , mkF $ binop $    numOp    (-))
            , ("<"     , mkF $ binop $    numCmp   (<))
            , ("<="    , mkF $ binop $    numCmp   (<=))
            , (">"     , mkF $ binop $    numCmp   (>))
            , (">="    , mkF $ binop $    numCmp   (>=))
            , ("=="    , mkF $ binop $    numCmp   (==))
            , ("even?" , mkF $ unop $     numBool   even)
            , ("odd?"  , mkF $ unop $     numBool   odd)
            , ("pos?"  , mkF $ unop $     numBool (< 0))
            , ("neg?"  , mkF $ unop $     numBool (> 0))
            , ("eq?"   , mkF $ binop   eqCmd )
            , ("bl-eq?", mkF  $ binop $ eqOp     (==))
            , ("and"   , mkF $ binopFold (eqOp     (&&)) (Bool True))
            , ("or"    , mkF $ binopFold (eqOp     (||)) (Bool False))
            , ("cons"  , mkF   Prim.cons)
            , ("cdr"   , mkF   Prim.cdr)
            , ("car"   , mkF   Prim.car)
            , ("file?" , mkF $ unop  fileExists)
            , ("slurp" , mkF $ unop  slurp)
            ]
```

Primitive Creation

Lets go through an individual example to see how all the types mash!

Function Definition

```
type Binary = LispVal -> LispVal -> Eval LispVal

(+) :: Num a => a -> a -> a

numOp :: (Integer -> Integer -> Integer) -> LispVal -> LispVal -> Eval LispVal

binopFold :: Binary -> LispVal -> [LispVal] -> Eval LispVal
```

Function Reduction

```
numOp (+) :: LispVal -> LispVal -> Eval LispVal
numOp (+) :: Binary

binopFold (numOp (+)) :: LispVal -> [LispVal] -> Eval LispVal
binopFold (numOp (+)) (Number 0) :: [LispVal] -> Eval LispVal

IFunc $ binopFold (numOp (+)) (Number 0) :: IFunc
mkF $ binopFold (numOp (+)) (Number 0) :: LispVal
```

Alright, so it's a complicated transformation, but as you can see the types do work out. The engineering principle at play here is the use of the function `numOp`, and similar functions, for as many operators as possible. This reduces the amount of code needed to be written. Further, `binop` and `unop` can be re-used for most functions. Varags would have to be handled differently, possibly by entering each pattern match individually.

Helper Functions

```
type Prim   = [(T.Text, LispVal)]
type Unary  = LispVal -> Eval LispVal
type Binary = LispVal -> LispVal -> Eval LispVal

unop :: Unary -> [LispVal] -> Eval LispVal
```

```haskell
unop op [x]    = op x
unop _ args    = throw $ NumArgs 1 args

binop :: Binary -> [LispVal] -> Eval LispVal
binop op [x,y]  = op x y
binop _  args   = throw $ NumArgs 2 args

binopFold :: Binary -> LispVal -> [LispVal] -> Eval LispVal
binopFold op farg args = case args of
                            [a,b]  -> op a b
                            (a:as) -> foldM op farg args
                            []-> throw $ NumArgs 2 args
```

So the `binop`, `unop`, and `binopFold` are basically unwrapping functions that take a `[LispVal]` and an operator and apply the arguments to the operator. `binopFold` just runs the `foldM`, while taking an additional argument. It should be noted that `binopFold` requires the operator to work over monoids.

IO Functions

```haskell
fileExists :: LispVal  -> Eval LispVal
fileExists (Atom atom)  = fileExists $ String atom
fileExists (String txt) = Bool <$> liftIO (doesFileExist $ T.unpack txt)
fileExists val  = throw $ TypeMismatch "expects str, got: " val

slurp :: LispVal  -> Eval LispVal
slurp (String txt) = liftIO $ wFileSlurp txt
slurp val          =  throw $ TypeMismatch "expects str, got:" val

wFileSlurp :: T.Text -> IO LispVal
wFileSlurp fileName = withFile (T.unpack fileName) ReadMode go
  where go = readTextFile fileName


readTextFile :: T.Text -> Handle -> IO LispVal
readTextFile fileName handle = do
  exists <- hIsEOF handle
  if exists
  then (TIO.hGetContents handle) >>= (return . String)
  else throw $ IOError $ T.concat [" file does not exits: ", fileName]
```

These are the basic file handling, `slurp`, which reads a file into a string, and `fileExists` which returns a boolean whether or not the file exists.

List Comprehension

```
cons :: [LispVal] -> Eval LispVal
cons [x,y@(List yList)] = return $ List $ x:yList
cons [c]                = return $ List [c]
cons []                 = return $ List []
cons _  = throw $ ExpectedList "cons, in second argumnet"

car :: [LispVal] -> Eval LispVal
car [List []    ] = return Nil
car [List (x:_)]  = return x
car []            = return Nil
car x             = throw $ ExpectedList "car"

cdr :: [LispVal] -> Eval LispVal
cdr [List (x:xs)] = return $ List xs
cdr [List []]     = return Nil
cdr []            = return Nil
cdr x             = throw $ ExpectedList "cdr"
```

Since the S-Expression is the central syntactical form of Scheme, list comprehension operators are a big part of the primitive environment. Ours are not using the `unop` or `binop` helper functions, since there are a few cases which varargs need to be support. A alternative approach would be to implement these as special forms, but since special forms are differentiated by their non-standard evaluation of arguments, they rightfully belong here, as primitives.

Unary and Binary Function Handlers

```
numBool :: (Integer -> Bool) -> LispVal -> Eval LispVal
numBool op (Number x) = return $ Bool $ op x
numBool op  x         = throw $ TypeMismatch "numeric op " x

numOp :: (Integer -> Integer -> Integer) -> LispVal -> LispVal -> Eval LispVal
numOp op (Number x) (Number y) = return $ Number $ op x  y
numOp op x          (Number y) = throw $ TypeMismatch "numeric op " x
numOp op (Number x)  y         = throw $ TypeMismatch "numeric op " y
numOp op x           y         = throw $ TypeMismatch "numeric op " x

strOp :: (T.Text -> T.Text -> T.Text) -> LispVal -> LispVal -> Eval LispVal
strOp op (String x) (String y) = return $ String $ op x y
strOp op x          (String y) = throw $ TypeMismatch "string op " x
strOp op (String x)  y         = throw $ TypeMismatch "string op " y
strOp op x           y         = throw $ TypeMismatch "string op " x

eqOp :: (Bool -> Bool -> Bool) -> LispVal -> LispVal -> Eval LispVal
eqOp op (Bool x) (Bool y) = return $ Bool $ op x y
eqOp op  x        (Bool y) = throw $ TypeMismatch "bool op " x
eqOp op (Bool x)  y        = throw $ TypeMismatch "bool op " y
eqOp op x         y        = throw $ TypeMismatch "bool op " x

numCmp :: (Integer -> Integer -> Bool) -> LispVal -> LispVal -> Eval LispVal
numCmp op (Number x) (Number y) = return . Bool $ op x  y
numCmp op x          (Number y) = throw $ TypeMismatch "numeric op " x
numCmp op (Number x)  y         = throw $ TypeMismatch "numeric op " y
numCmp op x         y           = throw $ TypeMismatch "numeric op " x


eqCmd :: LispVal -> LispVal -> Eval LispVal
eqCmd (Atom   x) (Atom   y) = return . Bool $ x == y
eqCmd (Number x) (Number y) = return . Bool $ x == y
eqCmd (String x) (String y) = return . Bool $ x == y
eqCmd (Bool   x) (Bool   y) = return . Bool $ x == y
eqCmd  Nil        Nil       = return $ Bool True
eqCmd  _          _         = return $ Bool False
```

These are the re-used helper functions for wrapping Haskell functions, and pattern matching the LispVal arguments. Further, the pattern matching can be used to dynamically dispatch the function at runtime depending on the data constructor of the arguments. This is a defining feature of dynamically typed programming languages, and one of the many reasons why their performance is slow compared to statically typed languages! Another key feature within these functions is the throwing of errors for incorrect types, or mismatching types, being passed to functions. This prevents type errors from being thrown in Haskell, and allows us to handle them in a way that allows for verbose error report. Example: `(+ 1 "a")` would give an error!

Conclusion

In summary, we make the primitive environment by wrapping a Haskell function with a helper function that pattern matches on `LispVals` and extracts the internal values that the Haskell function can accept. Next, we convert that function to be of type `[LispVal] -> Eval LispVal`, which is our type `IFunc`, the sole argument for the `LispVal` data constructor `Fun`. We can now map a `Text` value to a `LispVal` representing a function. This is our primitive environment, which should stay minimal, and if possible, new functions moved into the standard library if they can be defined from existing functions.

[Understanding Check]

Implement a new primitive function `nil?` which accepts a single argument, a list, returns true if the list is empty, and false under all other situations.
Use the `Lambda` constructor instead of `Fun` for one of the primitive functions.

Create a `Division` primitive function which works for `Number`, returning a `Number` by dividing then rounding down.
Write a new function,

```
binopFold1 :: Binary -> [LispVal] -> Eval LispVal
```

that uses the first value of the list as the identity element. More information, see here.

Read Eval Print Loop Repeat

Galileo alone had risked asserting the truth about our planet, and this made him a great man... His was a genuine career as I understand it. **Yevgeny Yevtushenko**

Repl.hs defines the code we use for our REPL loop. Our strategy will be to have the user enter text, parse, interpret, then display the result, then allow the user to enter another line of text. If an exception is thrown, we will catch and display the exception, then return to our normal mode.

```haskell
type Repl a = InputT IO a

mainLoop :: IO ()
mainLoop = runInputT defaultSettings repl
```

Here we define out `Repl` type using `InputT` from the mtl library to wrap `IO`. `mainLoop` will run the REPL with default settings, and is the top-level function exported by Repl.hs. that will be loaded into Main.hs and used to create the executable.

```haskell
repl :: Repl ()
repl = do
  minput <- getInputLine "Repl> "
  case minput of
    Nothing -> outputStrLn "Goodbye."
    Just input -> liftIO (process input) >> repl
    --Just input -> (liftIO $ processToAST input) >> repl
```

`repl` is a recursive function which will get a line of texted wrapped in the `Maybe` context, if `Nothing` then the function terminates, if `Just input` then we evaluate the text using the helper function `process`. Commented out is an alternative processing line, which will display the abstract syntax tree, very useful for debugging.

```haskell
process :: String -> IO ()
process str = do
  res <- safeExec $ evalText $ T.pack str
  either putStrLn return res

processToAST :: String -> IO ()
processToAST str = print $ runParseTest $ T.pack str
```

`process` takes a `String` input, packs it into `Text`, evaluates it, then handles errors using `safeExec`. Recall that `safeExec` return type `IO (Either String a)`. This catches thrown exceptions and allows for the REPL to display the error, then allow the user to subsequently enter a fixed expression.

Conclusion

The REPL here is pretty basic, nothing fancy, just connect the user to the underlying interpreter and let them play! It takes input, runs the evaluator, then displays either the computed `LispVal` or the `LispException`.

[Understanding Check]

Every input is evaluated independently, that is, with a fresh `EnvCtx`. Can you figure out a way to thread the `EnvCtx`, so variables bound in one inputted expression can be used in subsequent inputs?
REPL keywords like ":quit" ":help" ":clear" or ":ast" can be entered to allow for the REPL to do different tasks, like show the AST, information about commands, or quit out. Write a quick `Parsec` parser to extract these keywords.

Input/Output

Theirs not to make reply, theirs not to reason why, theirs but to do and die. **Alfred Tennyson**

Input Output

Evaluating S-Expressions as pure functions is conceptually simple: reduce terms, modify environments, and return the result. However, if usefulness is an aim, we must introduce side effects that model realworld interactions, namely reading and writing files. Haskell has already tackled this awkward squad via the `IO` monad, which we can use for our Scheme.

Working with IO inside Eval `Eval` is defined with `IO` inside our monad transformer stack:

```
newtype Eval a = Eval { unEval :: ReaderT EnvCtx IO a }
  deriving (Monad, Functor, Applicative, MonadReader EnvCtx,  MonadIO)
```

Which results in `EnvCtx -> IO a` monadic action. We also have the `liftIO` helper function:

```
liftIO :: MonadIO m => IO a -> m a
```

Thus, we have contained evaluation within an `IO` monadic action, and have a way, `liftIO`, to encapsulate `IO LispVal` into `Eval LispVal`. However, before we cover the functions provided for `IO`, there is still one important point, Exceptions.

Exceptions We covered exceptions and `safeExec` from Eval. hs in Chpater 4. To review. `safeExec` wraps runs a basic "try/catch" over the monadic action of evaluation to make sure exceptions are caught, and control resumed. Now that we are relatively 'safe' using `IO` inside evaluation, let's take a look at some of input and output functions.

IO Strategy: Goals and approach We are going to support two main operations: inputting in a script to execute, and reading and writing data files. We approach this by building smaller functions: primitives that compose well. To run scripts, we'll also need `parse` and `eval` functions to ingest the `String` values inputted from files. That's pretty much it, and we we are ready for our Haskell definitions.

Running A Script The first function we need for running a program within Scheme is called `slurp`, which takes a filename and returns a `String`. Within `slurp`, we are using `liftIO` to interleave `IO LispVal` and `Eval LispVal` actions. To avoid complete irresponsibility when reading files, `readTextFile` ensures the file being read actually exists, and if not, throws an informative `IOError` message.

```
slurp :: LispVal  -> Eval LispVal
slurp (String txt) = liftIO $ wFileSlurp txt
slurp val          =  throw $ TypeMismatch "read expects string, instead got: " val

wFileSlurp :: T.Text -> IO LispVal
wFileSlurp fileName = withFile (T.unpack fileName) ReadMode go
  where go = readTextFile fileName

readTextFile :: T.Text -> Handle -> IO LispVal
readTextFile fileName handle = do
  exists <- doesFileExist $ T.unpack fileName
  if exists
  then (TIO.hGetContents handle) >>= (return . String)
  else throw $ IOError $ T.concat [" file does not exits: ", fileName]
```

We are going to need a few more helper functions to run a program.
`parse` is defined in Eval. hs. For an inputed `String`, a `LispVal` representing the parsed structure is returned.

```
parseFn :: LispVal -> Eval LispVal
parseFn (String txt) = either (throw . PError . show) return $ readExpr txt
parseFn val = throw $ TypeMismatch "parse expects string, instead got: " val
```

Next, the `LispVal` can be evaluated using `eval`. We define a Scheme function `eval` in the primitive environment, which is just a shadowed version of the Haskell `eval` function defined throughout Eval. hs.
Putting all of this together we get:

```
(eval (parse (slurp "test/let.scm")))
```

which can be later defined as an entry in the standard library !

Reading and Writing Data Files We've covered reading files, lets take a look at writing to files. The situation is a bit complicated, as we need to use `String` to represent multiple types of `LispVal` if we want to store data. Fortunately, we have the `Show` typeclass for `LispVal` defined in a way that lets us parse/show `LispVal` values for all data constructors except `Fun` and `Lambda`. (See `showVal` in Prim.hs) Not being able to represent `Fun` and `Lambda` won't hold us back, as they only emerge to represent lambda functions during evaluation or primitive the primitive environment. Let's take a look at `put`, which follows the same structure as `slurp`.

```
put :: LispVal -> LispVal -> Eval LispVal
put (String file) (String msg) =  liftIO $ wFilePut file msg
put (String _)  val = throw $ TypeMismatch "put expects string in the second argument (try u
put val   _ = throw $ TypeMismatch "put expects string, instead got: " val

wFilePut :: T.Text -> T.Text -> IO LispVal
wFilePut fileName msg = withFile (T.unpack fileName) WriteMode go
  where go = putTextFile fileName msg

putTextFile :: T.Text -> T.Text -> Handle -> IO LispVal
putTextFile fileName msg handle = do
  canWrite <- hIsWritable handle
  if canWrite
  then (TIO.hPutStr handle msg) >> (return $ String msg)
  else throw $ IOError $ T.concat [" file does not exits: ", fileName]
```

There are a couple of differences besides arity, particularly `putTextFile` making a safety check via `hIsWritable`, and it should be noted that `put` returns the `String` value written. Looking at `put`, we see the second argument needs to be a `String`.
We define a primitive function `show`, defined as `Fun $ IFunc $ unop (return . String . showVal))`, taking advantage `showVal` defined in LispVal.hs.
Putting it all together we get:

```
Repl> (put "tmp1"  (show '(1 2 3)))
"(1 2 3)"
Repl> (parse (slurp "tmp1"))
"(1 2 3)"
Repl> (put "tmp1"  (show (+ 1 2 3)))
"6"
Repl> (parse (slurp "tmp1"))
"6"
```

Helper Functions Reading from a non-existent file is one of the most common and preventable IO mistakes, and a good demonstration of a helper function.

```
fileExists :: LispVal  -> Eval LispVal
fileExists (String txt) = Bool <$> liftIO (doesFileExist $ T.unpack txt)
fileExists val          = throw $ TypeMismatch "read expects string, instead got: " val
```

`fileExists` is a mapping of a single `LispVal` input into a function provided by System.Directory. For any serious use of our Scheme, it would be useful to wrap additional functions from System.Directory for file system manipulation.

wslurp: Files From The Web We can add functions from Network.Http as primtive function in our Scheme. A simple one, is an extended slurp that downloads websites.

```
openURL :: T.Text -> IO LispVal
openURL x = do
  req  <- simpleHTTP (getRequest $ T.unpack x)
  body <- getResponseBody req
  return $ String . T.pack body
```

```
wSlurp :: LispVal -> Eval LispVal
wSlurp (String txt) =  liftIO  $  openURL txt
wSlurp val = throw $ TypeMismatch "wSlurp expects a string, instead got: " val
```

This is another exmample of embedding `IO` into `Eval` monadic actions.

Conclusion Sincle we have `IO` within the monad transformer stack, we can use `liftIO` to perform than convert `IO a` to `Eval a` actions. This is important for reading and writing files, as well as other things, like foriegn function interfaces, or concurrency/parallel support. To prevent unchecked exceptions from crashing the REPL, we use the `safeExec` function, which runs actions within a try/catch block and displays the result. To read and write from files, we define our Scheme functions using a series of smaller, composable functions. Together, they can run scripts and read/write data files, and perform some basic checks on the file system.

[**Understanding Check**] Add a new Scheme function `appendTo` which appends its argument to a file.
Create some more helper functions from System.Directory, something like `rmFile`, `createDirectory`. If possible, abstract out as much of the interface as possible. One interesting addition would be a way to take a list of expressions, then execute each entry in a different thread, returning the results in a list.

Standard Library

Cry 'Havoc!', and let slip the dogs of war **William Shakespeare**

Standard Library

We define Scheme functions in three places: as special forms like `let`, in the primitive environment like the `+` operator, or within the body of an external Scheme program. The standard library is the third of these choices. Its functions are defined using building blocks of both primitive and special forms. If a function does not need a special evaluation strategy, or mapping to an internal operator, it belongs in the library. Successful programming languages will have a collection of functions that provide extensive capability and empower users to complete their tasks. When building a language, the standard library is often the first real "program" written in that language. When we write the library, we are not only providing the user with convenient shortcuts, but proving many features of the language work!

Our Standard Library We build the standard library, `test/stdlib_mod.scm` with a multitude of functions. Importantly, we allow recursive functions, like `fold` and `reduce` which enable efficient functional programming.

Composing `car` and `cdr` into `c({ad}^n)r` Our standard library defines functions that are combinations of `car` and `cdr`, for example `cdar` is `define cdar (lambda (pair) (cdr (car pair)))`. We can see these examples in the first defines of the standard library. For this to work, `car` and `cdr` must accept both quoted and unquoted lists. In other words, to chain these functions, they must accept S-Expressions without evaluating the arguments. This makes these functions specials forms.
In `Eval.hs` we add the special forms:

```haskell
eval all@(List [Atom "cdr", List [Atom "quote", List (x:xs)]]) =
  return $  List xs
eval all@(List [Atom "cdr", arg@(List (x:xs))]) =
  case x of
      Atom  _ -> do val <- eval arg
                    eval $ List [Atom "cdr", val]
      _           -> return $ List xs

eval all@(List [Atom "car", List [Atom "quote", List (x:xs)]]) =
  return $  x
eval all@(List [Atom "car", arg@(List (x:xs))]) =
  case x of
      Atom _       -> do val <- eval arg
```

```
                         eval $ List [Atom "car", val]
      _              -> return $ x
```

This feels like a hack, and if the `cadr` family of functions wasn't so essential, we could drop `car` and `cdr` as special forms.

Running Standard Library `Eval.hs` contains the functions that run text within the context of the standard library. Because the reader monad does not return the input context, we must wrap the expression and the standard library together as `LispVals`, then evaluate. This is done with the `endOfList` function.

Define the standard library file.

```
sTDLIB :: T.Text
sTDLIB = "lib/stdlib.scm"
```

Parse the input function as an S-Expression, then the library file as a list of S-Expressions. Append the parsed input expression to the end of the list.

```
endOfList :: LispVal -> LispVal -> LispVal
endOfList (List x) expr = List $ x ++ [expr]
endOfList n _  = throw $ TypeMismatch  "failure to get variable: " n


parseWithLib :: T.Text -> T.Text -> Either ParseError LispVal
parseWithLib std inp = do
  stdlib <- readExprFile std
  expr   <- readExpr inp
  return $ endOfList stdlib expr

getFileContents :: FilePath -> IO T.Text
getFileContents fname = do
  exists <- doesFileExist fname
  if exists then TIO.readFile  fname else return "File does not exist."
```

Final monadic evaluation of both standard library and expression. The key here is the `evalBody`, which accepts an S-Expression consisting of a series of `define` statements to be sequentially evaluated. This is the same evaluation strategy used in both `let` and `lambda` expressions.

```
textToEvalForm :: T.Text -> T.Text -> Eval LispVal
textToEvalForm std input = either (throw . PError . show )  evalBody $ parseWithLib std inpu

evalText :: T.Text -> IO () --REPL
evalText textExpr = do
  stdlib <- getFileContents $ T.unpack  sTDLIB
  res <- runASTinEnv basicEnv $ textToEvalForm stdlib textExpr
```

Conclusion

The standard library is the third and final place we define capability in our Scheme, after special forms and the primitive environment. The idea behind the standard library is that we have a relatively easy to write collection of utility and helper functions needed to get things done. If you look into the library, functions like `reduce`, `fold` and the `cadr` family are pretty useful.

However the `e -> a` functionality of `ReaderT` monadic action limits our approach. We cannot evaluate the file containing the library and get the modified environment back again. This somewhat complicates things, and requires us to take our approach via syntax manipulation. Although its not ideal, its also not very complex and lets us keep the simplistic lexical scoping via reader monad function `local` we established earlier. Alternatively, we can approach evaluation with `StateT` to run the monad, and get a modified `state`.

[**Understanding Check**] Add a new standard library function to generate the first 'n' Fibonacci numberers using a recursive function.
Define a pair of co-recursive functions, `even-co` and `odd-co`, that determine if a number is even or odd, in the standard library. `test/test_fix2.scm` contains a standard library that defines a Y-combinator. However, when we set this as the standard library then run the commented out expression, our expression fails to terminate. Why is this?
Create a new primitive `Vector` data type. Modify the parser to read `Vector` as numbers between brackets. In `Prim.hs` put the basic operations like add, multiply, then in standard library put operation like dot production, l2-distance. Include an auto-quote facility for the `Vector`special form.

Test

Testing shows the presence, not the absence of bugs **Dijkstra**

Testing w/ Haskell

Within dynamically typed language like Scheme, we lose the safety of Haskell's type system and need an alternative guaranty of behavior. This chapter is about writing tests to ensure our Scheme is behaving as we expect. Fortunately, testing can be easily integrated into a Stack project, and Haskell's many frameworks satisfy a multitude of testing requirements. We will be looking at a few testing options, and implementing both `HSpec`, and `Tasty.Golden`. There are two files for testing, test-hs/Spec.hs which contains the parser tests and golden file tests, and test-hs/Golden.hs, that contains just the golden file tests.

Haskell Testing Frameworks Haskell has a few good testing frameworks, here are a few of them, and what they do:
HUnit. A Simple embedded DSL for unit testing.
HSpec is a straightfoward testing framework, and gives great mileage for its complexity. HSpec is inspired by RSpec a testing library for Ruby, and the two show remarkable similarity.
QuickCheck. Tests properties of a program using randomly generated values.
Tasty Test framework that includes `HSpec`, `Quickcheck`, `HUnit`, and `SmallChcek`, as well as others, including `Golden`, which we will use for testing against a value within a file.

Testing Setup within Stack To setup testing, the following is added to `scheme.cabal`

```
test-Suite test
  type: exitcode-stdio-1.0
  main-is: Spec.hs
  hs-source-dirs: test-hs
  default-language: Haskell2010
  build-depends:
    base         >= 4.8 && < 5.0,
    text         >= 1.2 && <1.3,
    hspec        >= 2.2 && < 2.3,
```

This enables `stack test` and `stack build --test` to automatically run the `HSpec` tests found in test-hs/Spec.hs Running one of these commands will build the project, run the tests, and show the output. Phew! All tests pass! (let me know if they don't)

```
test-Suite test-golden
  type: exitcode-stdio-1.0
  main-is: Golden.hs
  hs-source-dirs: test-hs
  default-language: Haskell2010
  build-depends:
    base         >= 4.8 && < 5.0,
    text         >= 1.2 && <1.3,
    tasty        >= 0.11 && <0.12,
    tasty-golden >= 2.3 && <2.5,
    bytestring   >= 0.10.8 && <0.11,
    scheme == 0.1
```

Now we can run `stack test --test-golden` to run the tests from test-hs/Golden.hs Running `stack test` will perform both test suites. Let's look further into how testing is done, and the libraries used.

HSpec Setup `HSpec`'s strength is its simplicity, and ability to compare the results of arbitrary functions against a known output. We will use it to test internal components of our Scheme, particularly the parser, which contains a depth of logic orthogonal to the rest of the code base.
First, let's take a look at the general form of an `HSpec` test.

```haskell
main :: IO ()
main = do
  hspec $ describe "This is a block of tests" $ do
    it "Test 1" $
      textExpr input1 `shouldBe` "result of test 1"
    it "Test 2" $
      textExpr input2 `shouldBe` "result of test 1"
```

- `describe` gives a name to the block of tests, which is printed out when the tests are run.
- `it` sets a specific test.
- `shouldBe` states that a specific expression matches the test expression.

HSpec Tests Two internal aspects of our Scheme will be tested in test-hs/Spec.hs: The parser, and evaluation. These two features lend themselves easily to testing, and together, cover ensure functionality meets expectations.
Another view is that these tests allow us to modify the project without changing the features we worked so hard to implement, test driven development (TDD). The `./test` folder in our project contains the Scheme expressions run during the tests. Besides files containing expressions, we can also specify expressions as `T.Text`, and `define` blocks without loading the standard library. All of the parsing logic, and evaluation of simple expressions, special forms, and features like lexical scope are included in the scheme expressions found in the test folder.

Parser Tests The first set of tests ensures text is properly parsed into `LispVal` using `readExpr`. To organize this set of tests, the `hspec` function is used, along with `describe` to give the set of tests an suitable description. Many constructions of `LispVal` are tested, and here were divide that list into S-Expression and non-S-Expression values for simpler testing.

```
hspec $ describe "src/Parser.hs" $ do
  it "Atom" $
    readExpr "bb-8?" `shouldBe` (Right $ Atom "bb-8?")

  it "S-Expr: heterogenous list" $
    readExpr "(stromTrooper \"Fn\" 2 1 87)" `shouldBe`
      (Right $ List [Atom "stromTrooper", String "Fn", Number 2, Number 1,Number 87])
```

Eval Tests Alright, on to evaluation. Our task here is ensuring there are no errors, bugs, or unspecified behavior in our Scheme... If there were only a way to incorporate a system that protects us from invalid programs... Type systems be damned! *We are all that is Scheme!*
To test evaluation, we are going to either: read and parse from a file or inline text, then run with or without loading the standard library. This way, we have flexibility over testing conditions, especially considering the standard library will be subject to the majority of iterative testing and revision efforts.

```
hspec $ describe "src/Eval.hs" $ do
   wStd "test/add.scm"              $ Number 3
   wStd "test/if_alt.scm"           $ Number 2
   runExpr Nothing "test/define.scm"        $ Number 4
   runExpr Nothing "test/define_order.scm"  $ Number 42
```

This is all fine, but requires a lot of helper functions to work, specifically the following:

```
wStd :: T.Text -> LispVal -> SpecWith ()
wStd = runExpr (Just "test/stdlib_mod.scm")

-- run expr w/o stdLib
tExpr :: T.Text -> T.Text -> LispVal -> SpecWith ()
tExpr note expr val =
    it (T.unpack note) $ evalVal `shouldBe` val
    where evalVal = (unsafePerformIO $ runASTinEnv basicEnv $ fileToEvalForm "" expr)


runExpr :: Maybe T.Text -> T.Text -> LispVal -> SpecWith ()
runExpr  std file val =
    it (T.unpack file) $ evalVal  `shouldBe` val
    where evalVal = unsafePerformIO $ evalTextTest std file
```

```
evalTextTest :: Maybe T.Text -> T.Text -> IO LispVal --REPL
evalTextTest (Just stdlib) file= do
  stdlib <- getFileContents $ T.unpack  stdlib
  f      <- getFileContents $ T.unpack file
  runASTinEnv basicEnv $ textToEvalForm stdlib  f

evalTextTest Nothing file = do
  f <- getFileContents $ T.unpack file
  runASTinEnv basicEnv $ fileToEvalForm (T.unpack file) f
```

What's troublesome here is the use of `unsafePerformIO` to read file contents and shed the `IO` monad. Stepping back, we are coding a test within a specific file, evaluating it with or without the standard library, then comparing it to a value compiled into the testing file. If we can admit `HSpec` is good at testing internals like the parser, its also fair to say its not great at this process of "golden tests" for our Scheme language. Fortunately there is a better way that allows us to run a test Scheme file and compare the result against a 'golden' value in a stored file!

Tasty Golden Tests The package Tasty.Golden gives us a function:

```
goldenVsString :: TestName -- ^ test name
  -> FilePath -- ^ path to the «golden» file (the file that contains correct output)
  -> IO LBS.ByteString -- ^ action that returns a string
  -> TestTree -- ^ the test verifies that the returned string is the same as the golden fil
```

This allows us to run the tests located in ./test and compare the results to the likewise named files in ./test/ans.

Looking in test-hs/Golden.hs, we can see a drastic simplification compared to `HSpec`!

```
import Test.Tasty
import Test.Tasty.Golden
import qualified Data.ByteString.Lazy.Char8 as C

main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "Golden Tests"
  [   tastyGoldenRun "add"           "test/add.scm"              "test/ans/add.txt"
    , tastyGoldenRun "if/then"       "test/if_alt.scm"           "test/ans/if_alt.txt"
    , tastyGoldenRun "let"           "test/let.scm"              "test/ans/let.txt"
    ...
  ]

tastyGoldenRun :: TestName -> T.Text -> FilePath -> TestTree
tastyGoldenRun testName testFile correct = goldenVsString testName correct  (evalTextTest (
```

Where the `evalTextTest` function from `HSpec` is used again.

Conclusion

> *Once in a while you get shown the light in the strangest of places if you look at it right* **Jerry Garcia**

Yeahp, that's our Scheme! Now its time for yours!

What We Have Done In broad terms, we've built a programming language that's dynamically typed, runtime evaluated, and largely based on Scheme. We've extensively relied on Haskell abstractions: functors, monads, and their transformers and derivatives. Although we do handle input/output, our language is pretty simple and lacks optimization. However, we are in a very good position to quickly extend the language in in a couple of different ways.

Where To Go From Here The Scheme we've created can be extended in the following ways:

- Paired List `LispVal` data constructor
- Type System
- Build out library to create a useful DSL
- Compile to DSL to a stack language, C, or LLVM
- Submit a PR and improve this project!

In many ways, I think of this project as the spiritual predecessor to Stephen Diehl's **Write You A Haskell**. I think you're ready, but that's really up to you!
