MP5 - Scheme

Logistics

- revision: 1.5
- due: July 20, 2016 (part 1)
- due: July 27, 2016 (part 2)

This MP is broken into two parts, to be completed over two weeks. All of the code we'll be supplying you with is present in your `app/Main.hs`, but we have not fleshed out the entire `README.md` yet.

Changelog

1.1

- Fixed types of data-constructors `DefVal` and `Macro` in README.
- Clarify that you cannot use the `space` parser either for `whitespace`.
- Fix right-arrow -> left-arrow in explanation of `do ...` notation for `prompt` helper function.
- Add single-character tests to `identifier` parser.
- Add more parser tests.
- Clarify "reserved word" in `aForm` parser section.

1.2

- Fixed "quasi-quote" -> "unquote" in `anUnquote` parser section.
- Clarify that whitespace can happen in any of the quote short-forms.
- Mixed Lifters section into Runtime.
- Added explanation to Evaluation section.
- Added evaluation tests.
- Added some high-level overviews to the problem sets.

1.3

- Added explanation of quoting, unquoting, and quasiquoting.

1.4

- Added semantics for most of `eval`.

1.5

- Fixed `SymVal` -> `IntExp` typo in QuasiQuoting explanation.
- Clarify that `primEq` works with `ConsVal` as well.
- Fix bug in semantics for `VarExp` so that it says that `s |-> v` is in or not in the environment instead of just `s`
- Fix bug in semantics about applying the function held in a `PrimVal` to a list of arguments, not using it as a variadic function.
- Clean up semantics a bit
- Added examples to `eval`.
- Re-arranged `eval` problem-set for better flow.

Objectives

The objective for this MP is to build an interpreter for a small Lisp-like language called Scheme. You will use the Parsec parser we went over in class to build a parser, an evaluator to interpret the code, and a printer to show the output. Together these will form your repl.

This language will have the normal things you would expect in a Lisp-like language, such as functions, integers, and lists. You will also write a macro system and explore how to use it. Macros give you the ability to program your programming language, redefining it to be anything you want.

Goals

- Create the ADTs necessary to store a Scheme statement
- Learn how to use parser combinators (specifically the Parsec library)
- Understand the basic syntax of Scheme and how to evaluate programs in it
- Create an REPL for your interpreter which handles manipulating the environment based on inputs from the user

Further Reading

We'll be using the `Parsec` parser combinator Haskell library extensively for this MP. It would be good to read some of the documentation, and perhaps look at some examples of how to use it. We've been supplying you with the parsers for each MP up to this point, so you can also look at those and see how we have used the `Parsec` library.

The Haskell.org Parsec Page has good links to a few tutorials and examples. As is often the case with excellently documented Haskell libraries, the best resource is the source itself. In particular, the "Combinators" section of that page is quite useful.

Getting Started

Relevant Files

In the file `app/Main.hs`, you'll find all the code that we supply you. The file `test/Tests.hs` contains the tests.

Running Code

As usual, you have to `stack init` (you only need to do this once).

To run your code, start GHCi with `stack ghci` (make sure to load the `Main` module if `stack ghci` doesn't automatically). From here, you can test individual functions, or you can run the REPL by calling `main`. Note that the initial `$` and `>` are prompts.

```
$ stack ghci
 ... More Output ...
Prelude> :l Main
Ok, modules loaded: Main.
*Main> main
```

To run the REPL directly, build the executable with `stack build` and run it with `stack exec main`.

Testing Your Code

The test set is not complete as of revision 1.0. Some of the tests are implemented but most are not. Do not count on it for accurately telling you whether you can take the ML or not.

You are able to run the test-suite with `stack test`:

```
$ stack test
```

It will tell you which test-suites you pass, fail, and have exceptions on. To see an individual test-suite (so you can run the tests yourself by hand to see where the failure happens), look in the file `test/Spec.hs`.

It will also tell you whether you have passed enough of the tests to receive credit for the ML. If you do not pass 60% of the tests on the MP, you will not receive credit for the associated ML.

You can run individual test-sets by running `stack ghci` and loading the `Spec` module with `:l Spec`. Then you can run the tests (specified in `test/Tests.hs`) just by using the name of the test:

Look in the file `test/Tests.hs` to see which tests were run.

Given Code

There is not much given code this time around. We will be needing the `Parsec` library for the parser you'll write, along with the `HashMap` library for storing the runtime environment.

```
module Main where

import Prelude hiding (lookup)
import System.IO (hFlush, hPutStr, hPutStrLn, hGetLine, stdin, stdout)
import Text.ParserCombinators.Parsec hiding (Parser)
import Text.Parsec.Prim
import Data.Functor.Identity
import Data.HashMap.Strict (HashMap, fromList, lookup, insert, union, empty)
```

Problems (Part 1)

In the first part of the MP, we will focus on describing the data-type of our Scheme, as well as reading programs written in our language into data-structures of that data-type.

First, you will create Algebraic Data Types (ADTs) to represent the structure of programs in our language as well as the values our language can produce. A program written in Scheme be represented as one particular data-structure of the `Exp` type, and a value produced by our Scheme will be represented by another data-structure of the `Val` type.

We will also need to actually build the data-structure of a program from the string that a user provides (the program they write). This process is called parsing, and we will be using the parser-combinator library `Parsec` to do this.

Datatypes

You *must match* our data-constructor specification *exactly*. If you add extra data-constructors, omit some of them, or provide data-constructors which don't meet the specification, your code will likely not compile with our tests.

Environments

We will have a `runtime` environment which stores a map from variable names to values (type `Val`). We've provided a type-synonym here for it:

```
type Env = HashMap String Val
```

Expressions

First, you'll need to implement the data-types which store the Scheme expressions that are input.

```
data Exp = IntExp Integer
         deriving (Show, Eq)
```

We've given you the first data-constructor, `IntExp :: Integer -> Exp`. You'll need to provide the other two, `SymExp` and `SExp`.

- `SymExp :: String -> Exp`
- `SExp   :: [Exp]  -> Exp`

The data-constructor `SymExp` corresponds to a "symbol", which could be a variable name, some defined function's name, or a primitive operator name.

The data-constructor `SExp` corresponds to a Scheme *s-expression*, which is at the heart of how Scheme works. As you will see, *s-expressions* are used for nearly everything in Scheme.

Values

You'll also need to provide the data-constructors for the values we'll be using in the Scheme language. We've provided the first two for you, `IntVal` and `SymVal`. You must provide the rest.

```
data Val = IntVal Integer
         | SymVal String
```

The remaning data-constructors should have types:

- `ExnVal  :: String                 -> Val`
- `PrimVal :: ([Val] -> Val)         -> Val`
- `Closure :: [String] -> Exp -> Env -> Val`
- `DefVal  :: String -> Val          -> Val`
- `ConsVal :: Val -> Val             -> Val`
- `Macro   :: [String] -> Exp -> Env -> Val`

`ExnVal` will be used to hold an error message so that the user can be signaled that an error occurred. `PrimVal` is used to hold the definition of a primitive operator in our language (such as `+`).[1] `Closure` is used to store a function definition. `DefVal` is used signal the REPL that a new binding should be entered into the environment. `ConsVal` can be used for building both pairs and lists in our Scheme. `Macro` is used to store the definition of a Scheme macro (much like `Closure` stores the definition of a Scheme function).

You also need to provide a `Show` instance for the `Val` datatype so that we can pretty-print things of type `Val`.

`IntVal`: Give the string corresponding to the integer.

`SymVal`: Give the string as-is.

`ExnVal`: Given the string `"*** Scheme-Exception: "`, followed by the exception message, followed by `" ***"`.

[1] Scheme functions and primitives are *variadic*, which means they operate on lists of arguments. That is why `PrimVal` holds a function of type `[Val] -> Val` instead of `Val -> Val -> Val`.

PrimVal: Give the string `"*primitive*"`.

Closure: Give the string `"*closure*"`.

DefVal: Give the name of the variable being defined.

ConsVal: Give (in parenthesis) the list or tuple. If the list ends in `SymVal "nil"`, then it is a proper list, and should be output with spaces between the elements and an extra space at the end. If it does not end with `SymVal "nil"`, then it is a tuple and should be output with spaces between all elements except the last two, which should be output with a dot between them. For example:

```
ConsVal (IntVal 3) (ConsVal (SymVal "oeunt") (SymVal "nil"))
```

is a list because it ends with `SymVal "nil"`. Thus it should be output as

```
(3 oeunt )
```

But, the following is a tuple (because it doesn't end with `SymVal "nil"`):

```
ConsVal (IntVal 3) (ConsVal (SymVal "eohino") (SymVal "noethu"))
```

so it should be output as:

```
(3 eohino . noethu)
```

Macro: Give the string `"*macro*"`.

Parsing

You must implement the parser for this MP. Scheme is a simple language though, so it is not too difficult.[2]

We start out with a prettier type for the parsers:

```
type Parser = ParsecT String () Identity
```

This type synonym allows us to write the type `Parser Exp` if our parser will read a `String` and return an `Exp`. Similarly, we could write `Parser Int` if our parser will read a `String` and return an `Int`. The alternative would be to write the entire type synonym each time, eg. `ParsecT String () Identity Exp` or `ParsecT String () Identity Int`. That would be a pain and is uglier.

We are using the `Parsec` library to build parser combinators. You can use the function `parseWith :: Parser a -> String -> Either ParseError a` given below to test your parsers by hand.

```
parseWith :: Parser a -> String -> Either ParseError a
parseWith parser input = parse parser "" input
```

For example, you can test if the `identifier :: Parser String` parser you have to write works:

```
*Main> parseWith identifier "eotnah"
Right "eotnah"
*Main> parseWith identifier "2222natuheant"
Left (line 1, column 1):
unexpected "2"
```

Lexicals

Lexical parsers are useful for chunking up the input stream into tokens which may have meaning in our programming language. These parsers will be of type `Parser Char` or `Parser String` because they haven't added any new type information to the underlying input stream. Later, we'll use grammatical parsers to take

[2]Scheme being a simple language to parse does not mean it's not powerful. It has exactly the right simplicity to allow very elegant and compact specifications.

these series of tokens and try to assign more information to them (by actually producing things of type `Exp` for example).

We've provided two lexical parsers for you, `adigit`, and `digits`. The give you a simple feel for how to use parser combinators to build more complex parsers from simple ones.

We encourage you to look at the Parsec documentation to see what functions like

- `oneOf :: Stream s m Char => [Char] -> ParsecT s u m Char`
- `many1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m [a]`

look like. Those are not all the combinators you will need for this MP, so you will need to at least glance through the documentation.

Remember that we have type-synonyms for the `ParsecT` type, so the above types could be more simply read as

- `oneOf :: [Char] -> Parser Char`
- `many1 :: Parser a -> Parser [a]`.

```
adigit :: Parser Char
adigit = oneOf ['0'..'9']

digits :: Parser String
digits = many1 adigit
```

Whitespace parser

You should define the `whitespace :: Parser String` lexical parser. All it does is consume any whitespace it sees (white space being defined as spaces, newlines, and tabs: `" \n\t"`). After each of your other parsers, it is useful to call the `whitespace` parser which will get rid of empty spaces at the beginning of the input so that the rest of the input can be parsed.

Do not use the spaces or space parser that Parsec provides. Would you learn anything doing that?

```
*Main> parseWith whitespace ""
Right ""
*Main> parseWith whitespace "\n\t  "
Right "\n\t  "
*Main> parseWith whitespace "\n   \t  "
Right "\n   \t  "
*Main> parseWith whitespace "\n   \t  eontu"
Right "\n   \t  "
*Main> parseWith whitespace "\n   \t  eontu   \n\t\t"
Right "\n   \t  "
*Main> parseWith whitespace "oenth\n   \t  eontu   \n\t\t"
Right ""
```

Identifier parser

Identifiers represent symbols and variable names. The first character of an identifier must be one of the set `"-*+/:'?><=!"`, plus the upper and lower-case letters. The remaining characters can be from this set combined with the digits.

Define a parser for the first character of the identifiers (defined in the paragraph above), and let's call it `identFirst`. Also define one for the following characters of the identifiers, `identRest`. Use `identFirst` and `identRest` to build up the `identifier` parser.

```
*Main> parseWith identFirst "oenuth3oen"
Right 'o'
*Main> parseWith identFirst "4oenuth3oen"
Left (line 1, column 1):
unexpected "4"
*Main> parseWith identFirst "'?eu"
Right '\''
*Main> parseWith identFirst "=='?eu"
Right '='
*Main> parseWith identRest "=='?eu"
Right '='
*Main> parseWith identRest "3=='?eu"
Right '3'
*Main> parseWith identifier "3=='?eu"
Left (line 1, column 1):
unexpected "3"
*Main> parseWith identifier "a3=='?eu"
Right "a3=='?eu"
*Main>
```

Grammaticals

Grammatical parsers add information relevant to our programming language to the input stream (such as type information). While a programs' tokens all may be valid (so that it passes the lexer), the way they are arranged into a program may not be valid (it may not pass the parser).

We've provided a very simple grammatical parser for you, `anInt`. It uses the `digits :: Parser String` parser to get a string of digits, and uses `read` on the result to turn it into an actual `Integer`. Then it is wrapped in an `IntExp` so that it is of type `Exp`. In this way, we can take meaningless strings (like `"324234"`) and give them meaning in our programming languages (eg. `IntExp 324234`).

```
anInt :: Parser Exp
anInt = do d <- digits
           return $ IntExp (read d)
```

Parsing symbols

Identifiers are stored in the `SymExp` data-constructor, and we will call them *symbols*. In Scheme, a symbol serves two purposes. As in most languages, it can represent a variable. The evaluator will have an `env` parameter that allows us to look up variables' values. A symbol can also be a value in its own right, which we're not handling until later.

Make a *grammatical* parser `aSym` which uses the lexical parser `identifier` to parse a symbol and return it as something of type `Exp`.

```
*Main> parseWith aSym "oenuth3oen"
Right (SymExp "oenuth3oen")
*Main> parseWith aSym "a3=='?eu"
Right (SymExp "a3=='?eu")
*Main> parseWith aSym "3=='?eu"
Left (line 1, column 1):
unexpected "3"
*Main>
```

Parsing forms

In Scheme, a *form* starts with a parenthesis and a name (an identifier), then some arguments, and then a closing parenthesis. Forms are used for everything in this language. If the initial symbol in a form is not a reserved word (a built-in form), it is taken to be a function name or primitive operator. For the rest of this MP, when we say "form", we are specifically referring to an s-expression with a specific symbol in the first position. Note also that you do not have to distinguish between reserved words, primitives, and defined function names at the parser; the evaluator will handle distinguishing them.

Update your parser to handle forms by adding the `aForm` parser. You will need to read an opening parenthesis, then a list of expressions, then a closing parenthesis. Internally, this structure is called an *s-expression*. We will use an `SExp` data-constructor (of type `[Exp] -> Exp`) to store them. Remember to consume all the extra whitespace which may occur anywhere inside a form!

Hint: Looking ahead a bit, you'll see the `anExp` parser which is for parsing expressions. Make sure you use that parser if you ever need to parse an expression (which you need to do for `aForm`).

If you've implemented this correctly, you should be able to run the following code (notice we are testing the parser directly; we are *not* inside the repl).

```
*Main> parseWith aForm "(f 10 30 x)"
Right (SExp [SymExp "f",IntExp 10,IntExp 30,SymExp "x"])
*Main> parseWith aForm "(  '?onetuh 5 a 2neotuah ++ taunh   )"
Right (SExp [ SExp [SymExp "quote",SymExp "?onetuh"]
            , IntExp 5
            , SymExp "a"
            , IntExp 2
            , SymExp "neotuah"
            , SymExp "++"
            , SymExp "taunh"
            ]
      )
```

Quotes, Quasi-Quotes, and UnQuotes

The quote operator tells Scheme to convert the next expression to a value, as a symbol or a list. You can quote anything in Scheme.

There are two ways to quote something in Scheme, and you should supply both. The long way is the special form `quote`. The shortcut way is the `'` operator. It is often used as a shortcut for `(list ...)` (which we will define soon) but it quotes all the arguments before they are evaluated.

The parser does not have to handle the long way, because that will already be handled correctly if you are parsing forms correctly. However, the parser will need to handle the short-form explicitly.

Add a parser `aQuote` to handle the `'` operator; any expression `'e` should be parsed as `SExp [SymExp "quote", e]`. Remember to use the `anExp` parser if you want to parse any expressions - also realize that this parser won't work until you finish the `anExp` parser.

A quasi-quote in Scheme is like a quote that can be escaped by an unquote. The long form for a quasi-quote is `(quasiquote <exp>)`, and will be handled correctly by your `aForm` parser already. But you need to handle the short-form, `` ` <exp> ``, so that `` `e `` should be parsed as `SExp [SymExp "quasiquote", e]`. Write the parser `aQQuote` which parses the short-form of a quasi-quote.

You'll also need to handle un-quotes (which will be semantically paired with quasi-quotes). The long form for unquotes is `(unquote <exp>)`, and the short form is `, <exp>` (with a comma). Once again, you don't need to handle the long form explicitly, just the short form. Write the parser `anUnquote` which parses the short-form of an un-quote.

Note that whitespace can occur between any of the quoters/unquoters and their respective expressions. Make sure you handle it.

Hint: You may find it more compact to write a parser `mkQuote :: Char -> String -> Parser Exp` which can be used to make the `aQuote`, `aQQuote`, and `anUnquote` parsers. This isn't necessary.

Finally, make the `anyQuote` parser, which will parse any of the quoted forms. This parser should parse `aQuote` *or* `aQQuote` *or* `anUnquote`.

Expression Parser

You need to make the top-level expression parser. The expression parser should be able to parse any type of expression, including `anInt`, `aSym`, `aForm`, or `anyQuote`. Note that **the order you try parsers in matters**. For good measure, make sure you handle any whitespace before the expression or after the expression. This way, anytime you use the `anExp` parser elsewhere, you also know it will handle any whitespace around the expression.

Problems (Part 2)

In part 1 we made a data-type to hold programs written in our language, as well as a parser to read programs (as a `String`) written in our language into a format that Haskell can understand (as an `Exp`). We need three more things to have a full-blown interpreter for our language: an environment, evaluation, and a REPL (read-eval-print-loop).

The environment will be used to store a map from variable names to values for future reference. We will be able to store integers and symbols, but also function definitions, lists, primitive operations, macros, and exceptions (anything of type `Val`). Our initial environment will just contain some primitive operations that a user of our language will have access to immediately on starting the REPL.

Evaluation is the process of turning programs in some programming language into values. Sometimes the expressions used will refer to variables which have been defined in our environment - in those cases the evaluator will be responsible to looking up the definition of the variable and using it appropriately.

Finally, the REPL will repeatedly ask the user for input and parse the input into a program ("read"), evaluate the program into a value ("eval"), print the value ("print"), and finally recursively call the REPL ("loop").

Environment

The constant `runtime` is the initial runtime environment for the repl; it is a map from `String` (identifiers) to `Val` (values). This will be used to hold the values of defined constants, operators, and functions. You will call `repl` with this `runtime` when running it.[3]

You need to initialize `runtime` with predefined primitive operators as well. This will make these operators available to users of your language.

```
runtime :: Env
runtime = foldl union empty [ runtimeArith
                            , runtimeComp
                            , runtimeBool
                            , runtimeUnary
                            , runtimeOther
                            ]
```

[3]You can call `repl` with any initial environment. `runtime` will just be a *default* initial environment to use.

We have provided some translators to go between Scheme values and Haskell values. They are `liftbool`, `lowerbool`, `liftint`, and `lowerint`. These can help when defining the various operator lifters.

```
liftbool :: Bool -> Val
liftbool False = SymVal "nil"
liftbool True  = SymVal "t"

lowerbool :: Val -> Bool
lowerbool (SymVal "nil") = False
lowerbool _              = True

liftint :: Integer -> Val
liftint = IntVal

lowerint :: Val -> Integer
lowerint (IntVal i) = i
lowerint _          = error "Cannot lower, not an IntVal!"
```

You can test your `runtime` using the provided function `withRuntime`:

```
withRuntime :: String -> String
withRuntime input
    = let (SExp (SymExp op : args)) = (\(Right r) -> r) . parseWith aForm $ input
      in  case lookup op runtime of
            Just (PrimVal f) -> show . f . map (flip eval runtime) $ args
            _                -> error $ "Failed lookup '" ++ show op ++ "' in 'runtime'."
```

Note that you need to provide the definition of `eval` (see the Evaluation section) for `IntExp` and `SymExp` before this function will work for most of these examples.

Arithmetic Operators

`liftIntOp` takes an operator and a base-case. If the supplied list of values is empty, then the base-case (lifted into the Scheme world) is used. If it's non-empty, then the operator is applied between all the elements of the list.

We've provided the first element of `runtimeArith`, for the addition operator (`+`). Add integer subtraction (`-`) and multiplication (`*`) to your `runtime` environment. You should use `liftIntOp` for these.

For example (note that this example is written in Scheme):

```
*Main> withRuntime "(+ 3 4 5)"
"12"
*Main> withRuntime "(- 3 4 5)"
"-6"
*Main> withRuntime "(-)"
"0"
*Main> withRuntime "(+)"
"0"
*Main> withRuntime "(* 3 5 9)"
"135"
*Main>
```

Boolean Operators

Add boolean operators `and` and `or` to your `runtime environment. You should use`liftBoolOp‘ for these.

`liftBoolOp` should take an operator that works on lists of `Bool` and make it work on list of `Val`. Then we can use it to make primitive boolean operators in Scheme out of Haskell boolean operators.

Note that you'll need `eval` working on at least the quote form for these examples and tests to work.

```
*Main> withRuntime "(and t t t nil)"
"t"
*Main> withRuntime "(and t t t 'nil)"
"nil"
*Main> withRuntime "(and)"
"t"
*Main> withRuntime "(or)"
"nil"
*Main> withRuntime "(or t 't)"
"t"
*Main> withRuntime "(or t 't nil)"
"t"
*Main> withRuntime "(or t 't nil 'nil)"
"t"
*Main> withRuntime "(or 'nil 'nil 'nil 'nil)"
"nil"
*Main> withRuntime "(and 3 2 5)"
"t"
*Main> withRuntime "(and 3 2 5 'nil)"
"nil"
*Main> withRuntime "(or 3 2 5)"
"t"
```

Comparison Operators

Add integer comparison operators to your `runtime` environment:

- `>`: Integer greater than
- `<`: Integer less than
- `>=`: Integer greater than or equal
- `<=`: Integer less than or equal
- `=`: Integer equal
- `!=`: Integer not equal

You should use `liftCompOp` for these. `liftCompOp` takes an integer comparison function in Haskell and lifts it to a variadic Scheme comparison operator. If the list is empty it should return Scheme's `True`. If the list is larger, it should compare the elements of the list pair-wise using the given operator and then logically `and` all of those together.

```
*Main> withRuntime "(< 3 4 5)"
"t"
*Main> withRuntime "(= 3 3 2)"
"nil"
*Main> withRuntime "(>= 3 4 2)"
"nil"
*Main> withRuntime "(>=)"
"t"
*Main> withRuntime "(>= 7)"
"t"
*Main>
```

Unary Operators

We have three unary primitive operators in our Scheme:

- `not`: Boolean not (only operates on first element)
- `car`: Extract first element of a cons cell
- `cdr`: Extract second element of a cons cell

Write a function `liftUnary` which applies the given unary operator to a single element of an input list, otherwise producing an `ExnVal` with the error message "`opName` is a unary operator". Then use `liftUnary` to define the above operators. Use `primCar` and `primCdr` as the unary functions for `car` and `cdr`. These should return an `Exnval` if the input is not a cons-cell.

Note: Revision 1.1 and prior called the function `primUnary` instead of `liftUnary`. You can name it either way, only `runtimeUnary` will be tested.

```
*Main> withRuntime "(not)"
"*** Scheme-Exception: `not` is a unary operator. ***"
*Main> withRuntime "(car)"
"*** Scheme-Exception: `car` is a unary operator. ***"
*Main> withRuntime "(cdr)"
"*** Scheme-Exception: `cdr` is a unary operator. ***"
*Main> withRuntime "(not 't)"
"nil"
*Main> withRuntime "(not 'nil)"
"t"
*Main> withRuntime "(not nil)"
"nil"
*Main> withRuntime "(not nil 't)"
"*** Scheme-Exception: `not` is a unary operator. ***"
*Main> withRuntime "(not nil 't 3)"
"*** Scheme-Exception: `not` is a unary operator. ***"
*Main> withRuntime "(not nil 3)"
"*** Scheme-Exception: `not` is a unary operator. ***"
*Main> withRuntime "(not 3)"
"nil"
*Main> withRuntime "(car '(3 5) )"
"3"
*Main> withRuntime "(car 3 )"
"*** Scheme-Exception: Not a cons cell: 3 ***"
*Main> withRuntime "(cdr '(3 7) )"
"(7 )"
```

Other Operators

There are two more primitive operators to define.

- `eq?`: Integer and symbol equality (including integers and symbols inside nested lists)
- `list`: Construct a cons-list from arguments

Write the function `primEq` which will check if a list of `Val` are equal. For an empty list or a singleton list, it should return Scheme's `True`. For anything else it should check that all elements of the list are the same `Val`.

Define functions `liftList :: [Val] -> Val` and `lowerList :: Val -> [Val]` `liftList` should take a list of `Val` and turn it into a proper Scheme cons-list. `lowerList` should take a proper Scheme cons-list and turn it into a Haskell list of `Val`. These two functions should have the property that `lowerList . liftList = id`. If a non-proper cons-list is lowered by `lowerList`, produce a Haskell `error`.

The `list` primitive operator form can just use the `liftList` function you defined before. This makes a Scheme list out of its arguments.

Evaluation

Evaluation is where a Scheme expression is turned into a Scheme value. This is where we actually assign "meaning" to programs in our programming language. We can think (roughly) of the parser being type `String -> Exp`, and the evaluator being type `Exp -> Val`. Except we also want to keep track of variable bindings the user might want to set, so we add in an environment to the evaluator, getting `Exp -> Env -> Val`.

Check parameter names

There are a few forms which define functions, which means they must accept parameters. See the `(define (params) body)` form below for an example. We want to rule out cases where the `params` are not just strings (`SymExp`). So we'll make a helper function `paramStrs` which returns `Either String [String]`, with `Left msg` meaning an error in the parameter list, and `Right [String]` giving the valid parameter names back.

```
*Main> paramStrs []
Right []
*Main> paramStrs [SymExp "x", SymExp "oentuh", SymExp "yoeu"]
Right ["x","oentuh","yoeu"]
*Main> paramStrs [SymExp "x", SymExp "oentuh", IntExp 6, SymExp "yoeu"]
Left "Must use only `SymExp` for parameter names."
*Main> paramStrs [SymExp "x", SymExp "oentuh", SExp [], SymExp "yoeu"]
Left "Must use only `SymExp` for parameter names."
```

Quoting, Quasi-Quoting, and Unquoting

You may want to work on the `eval` function a bit before tackling quoting. These are helper functions which handle quoting, quasi-quoting, and unquoting scheme expressions and values.

The `quote` function turns an `Exp` into a `Val` without changing it, so that it can be "stored" unevaluated and unquoted later for evaluation. It should turn a `SymExp` into a `SymVal`, an `IntExp` into an `IntVal`, and an `SExp` into a `ConsVal` by quoting all of the elements of the `SExp` and using `liftList` on the result.

The `unquote` function is the opposite of the `quote` function, turning a `Val` back into the `Exp` that created it. They should obey the property that `unquote . quote = id`. A `SymVal` should be turned into a `SymExp`, and an `IntVal` should be turned into an `IntExp`. A `ConsVal` should be lowered, the resulting values unquoted, and then wrapped in an `SExp`. You don't need to handle any other cases.

The `quasiquote` operator is a bit more involved. While a quote protects a Scheme expression from evaluation as long as it's quoted, a quasiquote can be "escaped" using unquotes. But quasiquotes and unquotes can be nested, so to escape an expression fully you must unquote it as many times as it has been quasiquoted. This means you have to keep track of how many quasi-quotes you've been nested in (hence the extra `Int` argument). If you see an `(unquote exp)` form in the `quasiquote` function, the counter should be decremented. If the expression becomes fully unquoted, it should be evaluated in the given environment. If you see another `(quasiquote exp)` form though, the counter should be increased in the recursive call on `exp`. For a `SymExp` or an `IntExp`, `quasiquote` should behave the same as `quote`.

Evaluation - the function!

Now we finally have all the machinery necessary to build up the `eval` function. We'll have `eval` look at the `Exp` it recieves as input. If it is a `SymExp` or an `IntExp` it will be handled directly. If it is an `SExp`, we'll

first check if it matches one of the built-in forms of the language. If not, we'll assume it's either a primitive function or a user-defined function and look it up in the environment. If the environment lookup fails, we'll return an `ExnVal`.

Hint: You can match each of the predefined forms below with *one* top-level pattern each.

Integer, Symbol, and Empty Forms

Modify `eval` to handle integer and symbol expressions, as well as the empty form. Integer expressions should just be turned into integer values and returned. Symbol expressions should be looked up in the environment. If there is a value bound to the symbol in the environment, return that value. Otherwise, produce the `ExnVal "Symbol <symbol name> has no value."`. The empty form should just evaluate to Scheme `False`.

$$[\![i \mid \sigma]\!] \Downarrow i$$

$$[\![s \mid \sigma]\!] \Downarrow v \;\Big|\; (s \mapsto v) \in \sigma$$

$$[\![s \mid \sigma]\!] \Downarrow \texttt{ExnVal} \;\Big|\; (s \mapsto v) \notin \sigma$$

$$[\![() \mid \sigma]\!] \Downarrow \texttt{False}$$

Variable Definition Forms

Now we want to allow the user to define variables. The variable definition form is `(def var exp)` (an s-expression). When `eval` sees that form, return a `DefVal` with variable bound to the evaluated expression.

$$\frac{[\![e \mid \sigma]\!] \Downarrow v}{[\![(def\ x\ e) \mid \sigma]\!] \Downarrow \texttt{DefVal}\ x\ v}$$

```
scheme> (def x (+ 10 20))
x
scheme> x
30
scheme> y
*** Scheme-Exception: Symbol y has no value.
```

Function Definition and Lambda Function Forms

Add in the ability for the user to define functions. This has the form `(define f (params) body)`. The parameters, body, and new environment should be wrapped in a `Closure`, and the result bound in a `DefVal` for returning. You should use `paramStrs` on the `(params)` to make sure that they are a valid argument list, returning the error message in an `ExnVal` if not.

We also want the user to be able to build recursive functions, so make sure to add the resulting `DefVal` to the environment stored inside the `Closure` so that the function can call itself.

$$ps = (p_1 \cdots p_n) \qquad \sigma' = \sigma \cup \{f \mapsto \texttt{Closure}\ ps\ e\ \sigma'\}$$

$$[\![(define\ f\ ps\ e) \mid \sigma]\!] \Downarrow \texttt{DefVal}\ f\ (\texttt{Closure}\ ps\ e\ \sigma') \;\Big|\; valid(ps)$$

$$[\![(define\ f\ ps\ e) \mid \sigma]\!] \Downarrow \texttt{ExnVal} \mid \neg valid(ps)$$

Also add in the lambda-function form, `(lambda (params) body)`, which should evaluate to a `Closure`. Make sure you validate the `params` of the lambda form as well.

$$[\![(lambda\ ps\ e) \mid \sigma]\!] \Downarrow \texttt{Closure}\ ps\ e\ \sigma \mid valid(ps)$$

$$[\![(lambda\ ps\ e) \mid \sigma]\!] \Downarrow \texttt{ExnVal} \mid \neg valid(ps)$$

```
scheme> (def x 1)
x
scheme> (define inc (y) (+ y x))
inc
scheme> (inc 10)
11
scheme> (def x 2)
x
scheme> (inc 10)
11
scheme> (define add (x y) (+ x y))
add
scheme> (add 3 4)
7
scheme> (lambda (x) (+ x 10))
*closure*
scheme> ( (lambda (x) (+ x 10)) 20)
30
scheme> (define mkInc (x) (lambda (y) (+ x y)))
mkInc
scheme> (def i2 (mkInc 2))
i2
scheme> (i2 10)
12
scheme> (define fact (n) (cond ((< n 1) 1 't (* n (fact (- n 1))))))
fact
scheme> (fact 5)
120
```

Conditional Form

We should have some sort of if expression, because that's useful. Define the `(cond (c1 e1 ... cn en))` form. If `c1` is true, then `e1` is evaluated. If it's false the next condition should be tried. Return Scheme `False` if there are no conditions or no values.

$$[\![(cond\ ()) \mid \sigma]\!] \Downarrow \texttt{False} \qquad [\![(cond\ (c_1)) \mid \sigma]\!] \Downarrow \texttt{False}$$

$$\frac{[\![c_1 \mid \sigma]\!] \Downarrow \texttt{True} \qquad [\![e_1 \mid \sigma]\!] \Downarrow v_1}{[\![(cond\ (c_1\ e_1 \cdots c_n\ e_n)) \mid \sigma]\!] \Downarrow v_1}$$

$$\frac{[\![c_1 \mid \sigma]\!] \Downarrow \texttt{False} \qquad [\![(cond\ (c_2\ e_2 \cdots c_n\ e_n)) \mid \sigma]\!] \Downarrow v}{[\![(cond\ (c_1\ e_1 \cdots c_n\ e_n)) \mid \sigma]\!] \Downarrow v}$$

```
scheme> (cond ((> 4 3) 'a (> 4 2) 'b))
a
scheme> (cond ((< 4 3) 'a (> 4 2) 'b))
b
scheme> (cond ((< 4 3) 'a (< 4 2) 'b))
nil
```

Let Form

Define the `(let ((x1 e1) ... (xn en)) body)` form. The definitions made in `((x1 e1) ... (xn en))` should be added using *simultaneous assignment* to the environment that `body` is evaluated in. You'll need to check that the experssions being bound (the `(x1 e1) ... (xn en)`) are well-formed (they are a form with two entries, the first being a variable name).

$$\frac{[\![e_1 \mid \sigma]\!] \Downarrow v_1 \cdots [\![e_n \mid \sigma]\!] \Downarrow v_n \qquad [\![e \mid \sigma \cup \bigcup_{i=1}^{n} \{x_i \mapsto v_i\}]\!] \Downarrow v}{[\![(let\ ((x_1\ e_1) \cdots (x_n\ e_n))\ e) \mid \sigma]\!] \Downarrow v}$$

```
scheme> (let ((x 5) (y 10)) (+ x y))
15
scheme> (def x 20)
x
scheme> (def y 30)
y
scheme> (let ((x 11) (y 4)) (- (* x y) 2))
42
scheme> x
20
scheme> y
30
```

Cons Form

Make the `(cons car cdr)` form, which simply wraps the `car` and the `cdr` in a `ConsVal`.

$$\frac{[\![car \mid \sigma]\!] \Downarrow carv \qquad [\![cdr \mid \sigma]\!] \Downarrow cdrv}{[\![(cons\ car\ cdr) \mid \sigma]\!] \Downarrow \texttt{ConsVal}\ carv\ cdrv}$$

```
scheme> (cons 2 (cons 3 4))
(2 3 . 4)
scheme> (cons 2 (cons 3 (cons 4 'nil)))
(2 3 4 )
scheme> (list (> 3 4) 't 15 'nil (< 5 2 3 5) (cons 3 (cons 4 3)))
(nil t 15 nil nil (3 4 . 3) )
scheme> (car (cons 'a 'b))
a
scheme> (cdr (cons 'a 'b))
b
scheme> (car (list 'a 'b 'c))
a
scheme> (cdr (list 'a 'b 'c))
(b c )
scheme> (cdr (list 'a))
nil
scheme> (cdr 'a)
*** Scheme-Exception: Not a cons cell: a ***
```

Quoting, Quasi-Quoting, and Unquoting Forms

We can define `eval` over the `(quote exp)` and `(quasiquote exp)` forms by simply calling the helper functions you wrote above. If we see the `(unquote exp)` form though, something is wrong, as that means we're unquoting more than we have quasi-quoted, so generate an `ExnVal` saying so.

```
scheme> 'a
a
scheme> '5
5
scheme> (quote a)
a
scheme> '*first-val*
*first-val*
scheme> ''a
(quote a )
scheme> (car (quote (a b c)))
a
scheme> (car '(a b c))
a
scheme> (car ''(a b c))
quote
scheme> '(2 3 4)
(2 3 4 )
scheme> (list (+ 2 3))
(5 )
scheme> '( (+ 2 3))
((+ 2 3 ) )
scheme> '(+ 2 3)
(+ 2 3 )
scheme> (eval '(+ 1 2))
3
scheme> (eval ''(+ 1 2))
(+ 1 2 )
scheme> (eval (eval ''(+ 1 2)))
3
scheme> (def a '(+ x 1))
a
scheme> (def x 5)
x
scheme> (eval a)
6
scheme> (def a 5)
a
scheme> ``(+ ,,a 1)
(quasiquote (+ (unquote 5 ) 1 ) )
scheme> ``(+ ,,a ,a)
(quasiquote (+ (unquote 5 ) (unquote a ) ) )
scheme> `(+ a ,,a)
(+ a *** Scheme-Exception: Cannot `unquote` more than `quasiquote`. *** )
scheme> ``(+ a ,,a)
(quasiquote (+ a (unquote 5 ) ) )
scheme> (eval ``(+ ,,a 1))
(+ 5 1 )
scheme> (eval (eval ``(+ ,,a 1)))
```

```
6
```

Eval Form

Make the `(eval exp)` form, which assumes that its argument `exp` is a quoted expression. So it must evaluate its argument `exp` in the current environment (just to reduce it to a `Val`, which can be unquoted) then unquote the result and evaluate the resulting `Exp` one more time to get the final `Val`.

$$\frac{[\![e \mid \sigma]\!] \Downarrow v \qquad [\![unquote(v) \mid \sigma]\!] \Downarrow v'}{[\![(eval\ e) \mid \sigma]\!] \Downarrow v'}$$

Macro Form

Define the `(defmacro f (params) exp)` form which defines a `Macro`. A `Macro` is exactly like a function except for when we go to actually apply it to something (handled in the Application Form).

Make sure you use `paramStrs` on the `params` to check their validity as parameter names.

$$ps = (p_1 \cdots p_n) \qquad \sigma' = \sigma \cup \{f \mapsto \texttt{Macro}\ ps\ e\ \sigma'\}$$

$$[\![(defmacro\ f\ ps\ e) \mid \sigma]\!] \Downarrow \texttt{DefVal}\ f\ (\texttt{Macro}\ ps\ e\ \sigma') \mid valid(ps)$$

$$[\![(defmacro\ f\ ps\ e) \mid \sigma]\!] \Downarrow \texttt{ExnVal} \mid \neg valid(ps)$$

```
scheme> (defmacro if (con then else) `(cond (,con ,then 't ,else)))
if
scheme> (def a 5)
a
scheme> (if (> a 2) 10 20)
10
scheme> (if (< a 2) 10 20)
20
scheme> (define fact (n) (if (< n 1) 1 (* n (fact (- n 1)))))
fact
scheme> (fact 10)
3628800
scheme> (defmacro mkplus (e) (if (eq? (car e) '-) (cons '+ (cdr e)) e))
mkplus
scheme> (mkplus (- 5 4))
9
```

Application Form

If none of those forms were matched, then assume that the left-most part of the form is a function to be applied to the rest of the arguments of a form. The application form is `(f arg1 ... argn)`.

We will evaluate `f` to decide what it is. If it is a primitive, we must apply the primitive function held in the `PrimVal` to the arguments (which are obtained by evaluating all of `arg1 ... argn`).

If it is a closure, either it was a function defined using the `(define ...)` form, or using the `(lambda ...)` form. Either way, we must evaluate the arguments, bind them to the parameters of the closure, insert those bindings into the closure environment, then evaluate the body of the closure in the resulting environment.

If it is a macro, it's handled similarly to a closure, but now we know that we want to "protect" the arguments structure as actual Scheme expressions. To do that, we quote the arguments, then bind them to the parameters of the macro and add that to the macros environment. The body of the closure is evaluated in the new environment, which has the effect of placing the Scheme code from the arguments inside the macro. Then the result is unquoted (to "unfreeze" the arguments), and the whole expression is evaluated again. In this way, we can transport Scheme code into a function which arranges it programmatically using Scheme directly.

Finally, if the first argument evaluates to any other type of `Val`, just return the value directly.

$$\frac{[\![a_1 \mid \sigma]\!] \Downarrow v_1 \cdots [\![a_n \mid \sigma]\!] \Downarrow v_n}{[\![(f\ a_1 \cdots a_n) \mid \sigma]\!] \Downarrow p\ [v_1, \cdots, v_n]} \mid (f \mapsto \texttt{Primval}\ p) \in \sigma$$

$$\frac{[\![a_1 \mid \sigma]\!] \Downarrow v_1 \cdots [\![a_n \mid \sigma]\!] \Downarrow v_n \qquad [\![e \mid \sigma' \cup \bigcup_{i=1}^{l} \{p_i \mapsto v_i\}]\!] \Downarrow v}{[\![(f\ a_1 \cdots a_n) \mid \sigma]\!] \Downarrow v} \mid (f \mapsto \texttt{Closure}\ (p_1 \cdots p_m)\ e\ \sigma') \in \sigma \wedge l = min(n, m)$$

$$\frac{[\![e \mid \sigma' \cup \bigcup_{i=1}^{l} \{p_i \mapsto quote(a_i)\}]\!] \Downarrow v \qquad [\![unquote(v) \mid \sigma]\!] \Downarrow v'}{[\![(f\ a_1 \cdots a_n) \mid \sigma]\!] \Downarrow v'} \mid (f \mapsto \texttt{Macro}\ (p_1 \cdots p_m)\ e\ \sigma') \in \sigma \wedge l = min(n, m)$$

REPL

Generating next environment

You'll need to define the `nextEnv` function, which returns a new environment given a value. If the value is a `DefVal`, then `nextEnv` should insert the new definition into the environment. If the value is anything else, the original environment should be returned unchanged.

Writing the REPL

Next you'll be defining the `repl :: Env -> IO ()`. You can use the two helper functions below to assist in defining the `repl`.

```
prompt :: String -> IO String
prompt str = hPutStr stdout str >> hFlush stdout >> hGetLine stdin

printLn :: String -> IO ()
printLn str = hPutStrLn stdout str >> hFlush stdout
```

The first function, `prompt`, will print a given string then wait for one line of user input. You can access that line of input inside a `do` block using the left-arrow `<-`. The second function, `printLn`, will just print a string to the console. Both functions have been made to immediately print their output, instead of buffering it.

For example:

```
    = do i1 <- prompt "input 1: "
         printLn $ "You typed '" ++ i1 ++ "'."
         i2 <- prompt "input 2: "
         printLn $ "Your input doubled is: '" ++ i2 ++ i2 ++ "'."
```

could lead to the following interaction:

```
input 1: hello
You typed 'hello'.
input 2: world
Your input doubled is: 'worldworld'.
```

Your `repl` needs to prompt the use for input, and check if the input is equal to the string "quit". If so, then the `repl` should quit (by not recursing). If not, then parse the input using `parseWith`. If it passes the parser, then the resulting expression should be evaluated, printed, and `repl` should be called recursively in the potentially updated environment. If it doesn't pass the parser, the parse-error should be displayed and the `repl` called recursively in the current environment.

Main function

We've provided a `main` function for you, which just calls your `repl` with `runtime` as the initial environment.

```
main :: IO ()
main = do printLn "Welcome to your Scheme interpreter!"
          repl runtime
          printLn "Goodbye!"
```