MP5 - Scheme

Objectives

The objective for this MP is to build an interpreter for a minimalist dialect of Lisp called Scheme. You will learn to build a fully monadic evaluator, and a read-evaluate-print-loop (REPL) to accept input from user.

This language will have the normal things you would expect in a Lisp-like language, such as functions, numbers, symbols, 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

- Understand the basic syntax of Scheme and how to evaluate programs in it
- Understand how to simulate stateful computation by composing monads, and write seemingly imperative but under-the-hood functional code in Haskell
- Create a REPL for your interpreter which handles manipulating the environment based on inputs from the user
- Understand homoiconicity and metacircular evaluation via Scheme

Getting Started

Relevant Files

In the directory `app/`, you will find the program code, some of which is only partially implemented, and which you will have to modify to complete this assignment. The file `test/Tests.hs` contains the code used for testing.

Running Code

If you'd like to test your code on snippets of Scheme code, start GHCi with `stack ghci`. (Make sure to load the `Main` module if `stack ghci` doesn't automatically do it). From there you can run the REPL by calling `main`. For example:

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

(Note that the initial `$` and `>` are prompts, and not what you should type.)

You can run the REPL directly by building the executable with `stack build` and running it with `stack exec main`.

Testing Your Code

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`.

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

In directory `app/`:

- `Main.hs`: partially implemented REPL frontend

In directory `app/Scheme/`:

- `Core.hs`: fully implemented core language data structures
- `Parse.hs`: fully implemented parser
- `Eval.hs`: partially implemented evaluator
- `Runtime.hs`: partially implemented runtime routines

Environment

Like previous assignments, the environment is a `HashMap`. You can access functions like `lookup`, `union` and `insert` through prefix `H`, such as: `H.lookup`

```haskell
type Env = H.HashMap String Val
```

Abstract Syntax Tree

Now we offer you the Scheme AST. From your previous experience, do you notice anything unusual?

```haskell
data Val = Symbol String
         | Boolean Bool
         | Number Int
         | Nil
         | Pair Val Val
         | PrimFunc ([Val] -> EvalState Val)  -- Primitive function,
                                             --   implemented in Haskell
         | Func [String] Val Env             -- Closure
         | Macro [String] Val                -- Macro
         | Void                              -- No value
```

That's right, the AST is represented using values (`Val`) instead of expressions (`Exp`). Scheme (as well as other Lisp dialects) is a homoiconic language, meaning that the text of the language and the values are represented with the same data structure.

> Code is data. Data is code.

Expressions in the program are encoded as values: numbers, booleans, symbols and lists. When you feed such a value to the evaluator, the evaluator treats it as an expression and evaluates it!

Now we'll describe what the different values represent, and provide some examples.

Kinds of values

1. `Symbol`

   A Symbol is a string that represents a symbolic constant, such as a variable or function name. In previous MPs, we would represent a symbol as an expression, a.k.a. `SymExp` or `VarExp`. A reference to a `VarExp` would take the variable name, and evaluate it to the value it was assigned to. However in Scheme, it is also possible for the symbol itself to be used as a value.

2. `Boolean`

   Standard boolean value, can be either true `#t` or false `#f`.

3. `Number`

   For this MP, numbers are just integer values, we will not be implementing floating point values.

4. `Nil`

A special value used to represent the empty list.

5. `Pair`

   In Scheme, both pairs as well as lists are represented using the pair data structure. A pair is considered to be a list if the second element is also a list - either another list-pair, or the empty list Nil or `()`. In this way, pairs are used in the same manner as the "cons" operator in Haskell. We refer to such a pair as a "proper list".

   The syntax for a pair is `(1 . 2)`, however if a pair is a proper list, then it is printed with special list syntax. For instance, `(1 . (2 . (3 . () )))` would be written out at simply `(1 2 3)`.

   It's important to realize that for all of the syntactic forms we use in this MP, any list of expressions is actually a pair representing a "proper list", and ending with the `Nil` object.

6. `PrimFunc`

   A primitive function is a function defined in Haskell, lifted to Scheme. The type constructor `PrimFunc` takes a function that takes an argument list and returns an evaluation state, encapsulating either both the result of evaluation and the environment, or a `Diagnostic` thrown along the computation.

7. `Func`

   A closure has an argument list, a body, and a captured environment.

   As a side note, we are not implementing the reference memory model for Scheme. Thus functions are passed by value, i.e. all variables of the closure environment will be copied upon the copy of a closure.

8. `Macro`

   A macro has an argument list and a body. The body will be first transformed to an expanded body by the evaluator, and the expanded body gets fed back into the evaluator again. We'll talk about it in detail in the evaluation section.

9. `Void`

   The evaluator returns a value for every expression. Void is a special return type of the `(define ...)` and `(define-macro ...)` special forms. It does not represent any data.

Diagnostic

You are given a `Diagnostic` type which represents runtime errors thrown along evaluation. In the semantics you will be expected to throw certain `Diagnostic` at certain points of the evaluator.

```haskell
data Diagnostic = UnexpectedArgs [Val]
                | TypeError Val
                | NotFuncError Val
                | UndefSymbolError String
                | NotArgumentList Val
                | InvalidSpecialForm String Val
                | CannotApply Val [Val]
                | InvalidExpression Val
                | NotASymbol Val
                | NotAListOfTwo Val
                | UnquoteNotInQuasiquote Val
                | Unimplemented String
```

Evaluation State

At the end of `Scheme/Core.hs`, we defined for you the type of the evaluation state monad, `EvalState a`, where `a` is the type of the evaluation result.

```haskell
type EvalState a = StateT Env (Except Diagnostic) a
```

`StateT` is the monad transformer version of `State`. But you do not need to fully understand monad transformers! Simply read the declaration above as: `EvalState` is a state encapsulating the evaluation result of type `a` and the environment of type `Env`, except when a `Diagnostic` is thrown along the evaluation.

Unlike evaluators you have previously written in this course, the Scheme evaluator will *look like imperative code*. Under the hood, the `do` notation is doing function composition. The following example is part of the evaluator of the `define` special form for functions:

```haskell
do -- Save the current environment
  env <- get
  -- Create closure value
  val <- (\argVal -> Func argVal body env) <$> mapM getSym args
  -- Modify environment
  modify $ H.insert fname val
  -- Return void
  return Void
```

In order to work with the `EvalState` monad, you will use the following library functions. To explain briefly, because of how we defined our `EvalState` with `StateT`, `EvalState` is also an instance of some library typeclasses that provide us with these functions:

```haskell
-- Return the state from the internals of the monad.
get :: EvalState Env

-- Specify a new state to replace the state inside the monad.
```

```haskell
put :: Env -> EvalState ()

-- Monadic state transformer. Taking a function as its argument, it converts
-- the old state to a new state inside the state monad. The old state is lost.
modify :: (Env -> Env) -> EvalState ()

-- Used within a monadic computation to begin exception processing. We'll use
-- it to throw `Diagnostic` errors and still return an `EvalState`.
throwError :: Diagnostic -> EvalState a
```

Terminology

1. Value

   A value is just a `Val`. It's sometimes referred to as "datum".

2. Self-evaluating

   We call a datum self-evaluating if it always evaluates to itself. `Number` and `Boolean` are self-evaluating.

3. Form

   A form is a Scheme datum (`Val`) that is also a program, that is, it can be fed into the evaluator. It can be a self-evaluating value, a symbol, or a list.

4. Special form

   A special form is a form with special syntax and special evaluation rules, possibly manipulating the evaluation environment, control flow, or both.

5. Macro

   A macro is a form that stands for another form. An application of macro may look like a function application, but it goes through macro expansion first to get translated to the form it stands for, and then the expanded form will be evaluated.

6. Diagnostic

   A diagnostic is a run time error thrown along evaluation.

Problems

Caution

We recommend reading through the *entire* instructions PDF before beginning. Also, the notation may not appear correctly in the `.md` file, so please do read the PDF.

If you encounter an "unimplemented" error when evaluating a scheme expression in the examples, do not worry. It's up to you to go ahead and implement it, or keep following the order of the handout. You'll eventually implement these features, but you may have to go back and forth.

Execution

Problem 1. REPL

The first order of business is to implement the REPL, located in `Main.hs`. Besides our test suite, the REPL is what allows us to test snippets of Scheme code and to see how they evaluate.

The REPL is implemented as a function `repl :: Env -> IO ()`, and you'll have to implement certain cases of evalaution.

Our REPL starts by reading a string from the command line and then parsing it into the Scheme AST. If it fails to parse an error will be printed. Otherwise we will call a function `runExcept` returning type `Either Diagnostic (Val, Env)`.

The `Either` datatype is an error-handling datatype similar to `Maybe`, but rather than returning a `Just` value upon success, it will return a `Right` value, where `Right` contains a value and a new environment. Additionally rather than returning `Nothing` on failure, it will return a `Left` value, containing a `Diagnostic` error.

```haskell
repl :: Env -> IO ()
repl env = do
  putStr "scheme> "
  l <- getLine                                          -- Read
  case parse exprP "Expression" l of                    -- Parse
    Left err -> print err                               -- Diagnostics
    Right expr ->
      case runExcept $ runStateT (eval expr) env of     -- Eval
        Left err -> print err
        -- TODO:
        -- Insert line here: If return value is void,
        --                   loop with new env without printing
        -- Insert line here: Otherwise, print and loop with new env
```

```
        --
        -- The following line may be removed when you're done implementing
        --  the cases above:
        _ -> print "Error in Main.hs: Finish implementing repl"
  repl env                                              -- Loop with old env
```

Your job is to take what the call to `runExcept` returns and implement the TODO. If an error diagnostic message is returned, print the error. (This case has already been given.) Otherwise, check to see the value that was returned. If it was `Void`, then go back to the start of the loop using the new environment. Otherwise, print the value before looping.

We do not automatically test your REPL in the test cases! As a sanity check, the following should work even if you haven't done the rest of the MP yet. These inputs correspond to catching an error, evaluating to a void, and evaluating to some other value:

```
scheme> (define (f x))
Error: Invalid pattern in special form `define`: (define (f x))
scheme> (define (f x) (1))
scheme> `(3 . 3)
(3 . 3)
```

After you implement the remainder of the MP, all sorts of inputs should work:

```
scheme> (cons 'monad '(is just a monoid in the category of endofunctors))
(monad is just a monoid in the category of endofunctors)
```

Evaluation

Evaluator

Here we will write and test our evaluator.

Problem 2. Integer & Boolean, the self-evaluating primitives

`Integer` and `Boolean` evaluate to themselves. They are examples of expressions in "normal form". When an expression evaluates, the goal is to continually evaluate it further, until it reaches a normal form.

About the notation: when n is evaluated in environment σ, the result is n and the environment remains the same σ.

$$[\![n \mid \sigma]\!] \Downarrow \langle n \mid \sigma \rangle$$

$$[\![\texttt{\#t} \mid \sigma]\!] \Downarrow \langle \texttt{\#t} \mid \sigma \rangle$$

$$[\![\texttt{\#f} \mid \sigma]\!] \Downarrow \langle \texttt{\#f} \mid \sigma \rangle$$

Problem 3. Symbol

A `Symbol` evaluates to the value that it is bound to in the current environment.

If the value is not in the current environment, we throw an error instead. Note that for errors thrown, you have to throw the correct type of error. For the purposes of this assignment, you are free to pass any argument that you find useful as an argument to the error.

$$\frac{(s \mapsto v) \in \sigma}{[\![s \mid \sigma]\!] \Downarrow \langle v \mid \sigma \rangle}$$

$$\frac{(s \mapsto v) \notin \sigma}{[\![s \mid \sigma]\!] \Downarrow \langle \texttt{UndefSymbolError} \mid \sigma \rangle}$$

Problem 4. Special form `define` for variables

Now that we've given a way to reference variables, we want to allow the user to define variables. Variable definitions have the form `(define var exp)` (an s-expression), where `var` is a `Symbol`. The evaluator will evaluate the `exp` and bind the symbol to the resulting value.

Up to this point we've only been matching individual values. How do we match an s-expression? In our implementation s-expressions are implemented as lists of expressions. As we mentioned in our description of the AST, lists are built recursively using the `Pair` datatype.

In the skeleton of the code we've already written a case designed to handle all pairs / lists / s-expressions.

```
eval expr@(Pair v1 v2) = case flattenList expr of
  Left _ -> throwError $ InvalidExpression expr
  Right lst -> evalList lst where
```

Our code first "flattens" the pair to see if it is actually a proper list. If it is, we call `evalList`, which will check the structure of the list to see if it matches one of the language's special syntactic forms - in our case `(define var exp)`. Most of our language's syntax from here on out will be implemented as as cases of `evalList`.

You can use `put` or `modify` to mutate the state of the state monad. `put` takes an environment and set it as the new state, `modify` takes a function that performs an operation over environments and performs it on the state.

In our notation, σ is the original environment, σ' is an environment that *may* have been modified recursively, and finally we use substitution notation to show σ' gets updated with the new binding for x.

$$\frac{[\![e \mid \sigma]\!] \Downarrow \langle v \mid \sigma' \rangle}{[\![(\texttt{define}\ x\ e) \mid \sigma]\!] \Downarrow \langle \texttt{Void} \mid \sigma'[x \mapsto v] \rangle}$$

```
scheme> (define a (+ 10 20))
scheme> a
30
scheme> b
Error: Symbol b is undefined
```

(In *real* Scheme, you're not allowed to have a nested `define`, but our evaluator doesn't check for this, which allows for some strange program behavior.)

Problem 5a. Application Form (Closures)

Next we would like to be able to define not only variables, but also functions. However, before we can test function definitions, we have to be able to call functions.

Function application is the case when we have an s-expression, but none of the special syntactic forms are being used.

```
(f arg1 ... argn)
```

This is stubbed out as the final, wildcard case of `evalList`.

```
evalList (fexpr:args) =
  do f <- eval fexpr
     apply f args
```

The first expression is evaluated, and then we call the `apply` function, typed `apply :: Val -> [Val] -> EvalState Val`. The first argument is the function value, and the second argument are the (un-evaluated) expressions being passed to the function. This function's behavior is different based on whether the first argument is a primitive function, function closure, or a macro.

We've already implemented the case for primitive functions. We leave aside macros for a little bit later, your job is to implement the case for function closures. The semantics for function application can be defined as follows:

$$\frac{[\![f \mid \sigma]\!] \Downarrow \langle \texttt{Func}\ p_1 \cdots p_n\ e\ \sigma_f \mid \sigma' \rangle \qquad [\![e_1 \mid \sigma']\!] \Downarrow \langle v_1 \mid \sigma_1 \rangle\ \cdots\ [\![e_n \mid \sigma_{n-1}]\!] \Downarrow \langle v_n \mid \sigma_n \rangle \qquad [\![e \mid \bigcup_{i=1}^{n} \{p_i \mapsto v_i\} \cup \sigma_f \cup \sigma_n]\!] \Downarrow \langle v \mid \sigma'' \rangle}{[\![f\ e_1 \cdots e_n \mid \sigma]\!] \Downarrow \langle v \mid \sigma_n \rangle}$$

A high-level summary of the steps the complete implementation will perform:

1. Evaluate `f` to a function closure (done before `apply` is called).
2. Evaluate the argument expressions `e1` to `en` (possible modifying the environment).
3. Save the environment (Hint: use `get` to get the environment from the state)
4. Insert the values from the closure environment into the current environment.
5. Bind function parameter names to the argument expression values and insert them into the environment.
6. Evaluate the function body
7. Restore the environment we saved in step 3
8. Return the result of step 6

A note about recursion in Scheme: As you can see from the steps above, when functions are applied, they get evaluated in an environment that *combines* the bindings where the application occurs with the bindings stored in the closure. That means a function can easily refer to itself. In some other languages, this is not how recursion is implemented. What would happen if a function body could only be evaluated in the closure environment?

Problem 5b. Special form `define` for named functions

We've already implemented this case for you.

A function definition has the form `(define (f params) body)`. The parameters, body, and environment when the function is declared get wrapped into a `Func` value. It uses `get` to retrieve the environment from the state monad, and `modify` to mutate the state. A `Func` value is also a normal form.

The semantics for this can be given as follows. The notation $valid(p_1 \cdots p_n)$ means that parameters $p_1 \cdots p_n$ (such as might be labeled `x y z` in `f x y z`, for example) must be a proper list of `Symbol`s, implemented with type `List [Symbol]`.

$$\frac{valid(p_1 \cdots p_n)}{[\![(\texttt{define}\ (f\ p_1 \cdots p_n)\ e) \mid \sigma]\!] \Downarrow \langle \texttt{Void} \mid \sigma[f \mapsto \texttt{Func}\ p_1 \cdots p_n\ e\ \sigma] \rangle}$$

$$\frac{\neg\, valid(p_1 \cdots p_n)}{[\![(\texttt{define}\ (f\ p_1 \cdots p_n)\ e) \mid \sigma]\!] \Downarrow \langle \texttt{NotASymbol} \mid \sigma \rangle}$$

```
scheme> (define x 1)
scheme> (define (inc y) (+ y x))
scheme> inc
```

```
#<function:(λ (y) ...)>
scheme> (inc 10)
11
scheme> (define x 2)
scheme> (define (add x y) (+ x y))
scheme> (add 3 4)
7
scheme> (define (fact n) (cond ((< n 1) 1) (else (* n (fact (- n 1))))))
scheme> (fact 5)
120
```

Note that named functions in Scheme can be used recursively. This isn't because of the binding semantics given here, but because of the mechanism by which they are applied. There is more detail later in this document.

Problem 5c. Special form `lambda` for anonymous function expressions

Given the case for named functions, we also require that you implement the form for anonymous / lambda functions. The lambda function form is `(lambda (params) body)`, which evaluates to a `Func`. The lambda creates an anonymous function to be used as a value, and it does *not* automatically bind it to a name in the environment.

One thing of note is that this is the first special form that requires you to match a nested list, the list containing `params`. Recall from our explanation of the AST that lists in Scheme are represented not as an actual Haskell list, but as a Scheme value.

You can convert this value into a list using the `getList` function, which is present in `Eval.hs`. You may need to do this for other problems as well. Even after it's converted to a list, it will be a list of values. You will need to manipulate it further if you want it to become a list of strings.

$$\frac{valid(p_1 \cdots p_n)}{[\![(\texttt{lambda}\ (p_1 \cdots p_n)\ e) \mid \sigma]\!] \Downarrow \langle \texttt{Func}\ p_1 \cdots p_n\ e\ \sigma \mid \sigma \rangle}$$

$$\frac{\neg\, valid(p_1 \cdots p_n)}{[\![(\texttt{lambda}\ (p_1 \cdots p_n)\ e) \mid \sigma]\!] \Downarrow \langle \texttt{NotASymbol} \mid \sigma \rangle}$$

In usage, a lambda expression could be applied immediately where it is written:

```
scheme> (lambda (x) (+ x 10))
#<function:(λ (x) ...)>
scheme> ((lambda (x) (+ x 10)) 20)
30
```

Or, a lambda expression could be explicitly bound to a name in the environment using `define`. You can even parameterize it by supplying additional parameters with the `define`:

```
scheme> (define (incBy x) (lambda (y) (+ x y)))
scheme> (define i2 (incBy 2))
scheme> (i2 10)
12
```

This partially allows for currying. However, direct application of such functions still must obey the proper nesting of parentheses:

```
scheme> (incBy 2 10)
Error: Cannot apply #<function:(λ (x) ...)> on argument list (2 10)
scheme> ((incBy 2) 10)
12
```

Problem 6. Special form `cond`

Note: Several of the test cases for `cond` will not work until quotes are implemented in one of the later parts. If all the non-quote test cases work, you can move on. You could also

Instead of an if expression, Scheme makes use of the `cond` form, defined as `(cond (c1 e1) ... (cn en))`. If `c1` evaluates to true, then `e1` is evaluated and returned. If `c1` evaluates to false, then the next condition is evaluated.

The last condition, `cn`, can optionally be symbol `else`. The expression following `else` will be evaluated when all previous conditions evaluate to false. If `else` appears in one of the conditions that is not the last condition, it's an invalid special form (throw an error). If conditions are not exhaustive, i.e. when all conditions evaluate to false, return `Void`.

Note that you are given the function `getListOf2` in `Eval.hs` which can be used to verify that a Scheme list has length of two, and then returns a Haskell pair.

$$[\![(\texttt{cond}) \mid \sigma]\!] \Downarrow \langle \texttt{InvalidSpecialForm} \mid \sigma \rangle$$

$$\frac{[\![c_1 \mid \sigma]\!] \Downarrow \langle \texttt{Truthy} \mid \sigma' \rangle \qquad [\![e_1 \mid \sigma']\!] \Downarrow \langle v_1 \mid \sigma'' \rangle}{[\![(\texttt{cond}\ (c_1\ e_1) \cdots (c_n\ e_n)) \mid \sigma]\!] \Downarrow \langle v_1 \mid \sigma'' \rangle} \quad \text{where Truthy is any non-False value and } n \geq 1$$

$$\frac{[\![c_1 \mid \sigma]\!] \Downarrow \langle \texttt{False} \mid \sigma' \rangle \qquad [\![(\texttt{cond}\ (c_2\ e_2) \cdots (c_n\ e_n)) \mid \sigma']\!] \Downarrow \langle v \mid \sigma'' \rangle}{[\![(\texttt{cond}\ (c_1\ e_1) \cdots (c_n\ e_n)) \mid \sigma]\!] \Downarrow \langle v \mid \sigma'' \rangle} \quad \text{where } n \geq 2$$

$$\frac{[\![e \mid \sigma]\!] \Downarrow \langle v \mid \sigma' \rangle}{[\![(\texttt{cond}\ (\texttt{else}\ e)) \mid \sigma]\!] \Downarrow \langle v \mid \sigma' \rangle}$$

$$\frac{[\![c_1 \mid \sigma]\!] \Downarrow \langle \texttt{False} \mid \sigma' \rangle}{[\![(\texttt{cond}\ (c_1\ e_1)) \mid \sigma]\!] \Downarrow \langle \texttt{Void} \mid \sigma' \rangle}$$

$$[\![(\texttt{cond}\ (\texttt{else}\ e_1) \cdots (c_n\ e_n)) \mid \sigma]\!] \Downarrow \langle \texttt{InvalidSpecialForm} \mid \sigma \rangle$$

```
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))
(no output)
```

Problem 7. Special form `let`

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 expressions being bound (the `(x1 e1) ... (xn en)`) are well-formed (they are a form with two entries, the first being a variable name).

Note that in `Eval.hs` you are given a function `getBinding` which checks if a single $(x_i\ e_i)$ clause of a `let` or `let*` is a proper list where the first element is a symbol, which is evaluated to a `String`; then, it evaluates the second element to a `Val`, and returns a Haskell tuple of type `(String, Val)`.

$$\frac{[\![e_1 \mid \sigma]\!] \Downarrow \langle v_1 \mid \sigma_1 \rangle \cdots [\![e_n \mid \sigma]\!] \Downarrow \langle v_n \mid \sigma_n \rangle \qquad [\![e_{\text{body}} \mid \bigcup_{i=1}^{n} \{x_i \mapsto v_i\} \cup \sigma]\!] \Downarrow \langle v \mid \sigma' \rangle}{[\![(\texttt{let}\ ((x_1\ e_1) \cdots (x_n\ e_n))\ e_{\text{body}}) \mid \sigma]\!] \Downarrow \langle v \mid \sigma \rangle}$$

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

Problem 8. Special forms `quote`, `quasiquote` and `unquote`

Note: These cases are quite simple, however some of the test cases will not work until the assignment has been completed. You should be able to get the test cases that do not use `eval` completed however, and after this point conditionals should work as well.

By default all code that appears in a program will be evaluated to a value. The special forms `quote` and the related forms `quasiquote` and `unquote` so that some code can be used as data instead.

The special form `quote` returns its single argument, as written, without evaluating it. This provides a way to include constant symbols and lists, which are not self-evaluating objects, in a program.

The special form `quasiquote` allows you to quote a value, but selectively evaluate elements of that list. In the simplest case, it is identical to the special form `quote`. However, when there is a form of pattern `(unquote ...)` within a `quasiquote` context, the single argument of the `unquote` form gets evaluated.

As a homoiconic language, most of Scheme's syntax is exactly the same as the AST representation. There are, however, three special tokens in the *read syntax* (human-readable, to be desugared by the parser to Scheme AST as the internal representation) as a syntactic sugar for quoting, quasi-quoting and unquoting. You do not need to implement the desugaring since it's handled by our parser.

- `'form` is equivalent to `(quote form)`
- `` `form `` is equivalent to `(quasiquote form)`
- `,form` is equivalent to `(unquote form)`

```
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> (define a '(+ x 1))
scheme> (define x 5)
scheme> (eval a)
6
scheme> (define a 5)
scheme> ``(+ ,,a 1)
(quasiquote (+ (unquote 5) 1))
scheme> ``(+ ,,a ,a)
(quasiquote (+ (unquote 5) (unquote a)))
scheme> `(+ a ,,a)
Error: `unquote` not in a `quasiquote context: (unquote (unquote a))`
scheme> ``(+ a ,,a)
(quasiquote (+ a (unquote 5)))
scheme> (eval ``(+ ,,a 1))
(+ 5 1)
scheme> (eval (eval ``(+ ,,a 1)))
6
```

As for the semantics of these special forms, here's the semantics for quote:

$$[\!['\!e \mid \sigma]\!] \Downarrow \langle e \mid \sigma \rangle$$

And the semantics for a top-level unquote:

$$[\![,\!e \mid \sigma]\!] \Downarrow \langle \texttt{UnquoteInQuasiquote} \mid \sigma \rangle$$

The semantics for quasi-quote are significantly more complex, however we have already implemented this case for you, which you are free to look at if you would like insight into how quasiquotes work.

Problem 9a. Special form `define-macro` / Macro Application

Lastly, we get to write macro functionality. This problem has two parts, both of which must be completed before the test cases will pass.

Define the `(define-macro (f params) exp)` form which defines a `Macro`. A `Macro` is similar to a function: the key difference is not here in its definition binding, but later in its application, where we actually do evaluation twice. First, we evaluate the body of the macro, processing the arguments as frozen syntactic pieces *without evaluating them individually*, getting a new syntax blob. Then, we feed the result back into the evaluator to get the final result. In essence, macros use lazy evaluation.

$$ps = p_1 \cdots p_n$$

$$[\![(\texttt{define-macro}\ (f\ ps)\ e) \mid \sigma]\!] \Downarrow \langle \texttt{Void} \mid \sigma[f \mapsto \texttt{Macro}\ ps\ e] \rangle \quad \text{if } valid(ps)$$

$$[\![(\texttt{define-macro}\ (f\ ps)\ e) \mid \sigma]\!] \Downarrow \texttt{InvalidSpecialForm} \quad \text{if } \neg\, valid(ps)$$

In your evaluator skeleton, we implemented a special form `if` for you, but it's commented out. We do not need `if` as a special form because it can be defined as a macro using `cond`!

```
scheme> (define-macro (if con then else) `(cond (,con ,then) (else ,else)))
scheme> if
#<macro (con then else) ...>
scheme> (define a 5)
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)))))
scheme> (fact 10)
3628800
scheme> (define-macro (mkplus e) (if (eq? (car e) '-) (cons '+ (cdr e)) e))
scheme> (mkplus (- 5 4))
9
```

Problem 9b. Application Form (Macros)

Finally, we go back to the `apply` function to implement the macro case.

Recall that the main distinction between macros and non-macros is that a macro manipulates its arguments *at the syntax level* before actually evaluating them.

With that in mind, the semantics of macro application is similar to the semantics of function application.

$$\frac{[\![f \mid \sigma]\!] \Downarrow \langle \texttt{Macro}\ p_1 \cdots p_n\ e \mid \sigma' \rangle \qquad [\![e \mid \bigcup_{i=1}^{n} \{p_i \mapsto e_i\} \cup \sigma']\!] \Downarrow \langle e' \mid \sigma'' \rangle \qquad [\![e' \mid \sigma']\!] \Downarrow \langle v \mid \sigma_3 \rangle}{[\![f\ e_1 \cdots e_n \mid \sigma]\!] \Downarrow \langle v \mid \sigma_3 \rangle}$$

A high-level summary of the steps will be:

1. Evaluate the function to a macro (completed before `apply` is called)
2. Save the environment
3. Bind arguments (without evaluating them) to the parameters of the macro and insert them to the environment
4. Evaluate the macro body (i.e. expand the macro body)
5. Restore the environment we saved in step 2
6. Evaluate the expanded form (the result of step 3) and return it

Runtime Library

This assignment is almost complete. One notable thing that we haven't implemented at all are the primitive operations of the language. Arithmetic operations, etc. Our version of Scheme implements primitive operations by implementing them as regular functions.

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. The main call to `repl` should provide this `runtime` as the starting environment. (It is also possible to call `repl` with a different starting environment for experimental purposes.) You can test your `runtime` using the REPL, so implement the REPL first!

Our version of `runtime` has already been initialized with most of the predefined primitive operations. Your job will be to extend the runtime with a few extra operations that are useful in different contexts.

```
runtime :: Env
runtime = H.fromList [ ("+", liftIntVargOp (+) 0)
                     , ("-", liftIntVargOp (-) 0)
                     , ("and", liftBoolVargOp and)
                     , ("or", liftBoolVargOp or)
                     , ("cons", PrimFunc cons)
                     , ("append", PrimFunc append)
                     , ("symbol?", PrimFunc isSymbol)
                     , ("list?", PrimFunc isList)
                     ]
```

Note that `runtime` is used in the initial call by `main` in `app/Main.hs`, which we have been using to test our Scheme code.

```
main :: IO ()
main = repl runtime
```

Problem 10. Type Predicates (`symbol?`, `list?`, `pair?`, `number?`, `boolean?`, `null?`)

(Recall that a "predicate" is a function that returns a Boolean value.) These functions check whether a data element is of the corresponding type. They must take a single argument, but contained in a Haskell list of type `[Val]`, so that they are compatible with the AST constructor `PrimFunc`.

Within `app/Scheme/Runtime.hs`, your job is to extend the definition of the `runtime` constant to include these functions. The `symbol?` case has already been implemented for you to get you started. The total list of predicates which must be implemented are:

- `symbol?` Checks whether the input is a `Symbol`.
- `list?` Checks whether the input is a "proper list", aka either the empty list or a pair whose second element is a proper list.
- `pair?` Checks whether the input is a `Pair`.
- `null?` Checks for an empty list.
- `number?` Checks whether the input is a `Number`.
- `boolean?` Checks whether the input is a `Boolean`.

Note: The `list?` case is the most complicated one, so you can try to implement the others before implementing it.

```
scheme> (symbol? 'a)
#t
scheme> (symbol? 'b)
#t
scheme> (symbol?)
Error: Unexpected arguments ()
scheme> (symbol? 3)
#f
scheme> (list? '(3 5))
#t
scheme> (list? '())
#t
scheme> (list? '(3 . (6 . 7)))
#f
scheme> (list? '(3 5 . 6))
```

```
#f
scheme> (list? '(3 5 (6 . 7)))
#t
scheme> (list? 3)
#f
scheme> (list? 3 5)
Error: Unexpected arguments (3 5)
scheme> (list?)
Error: Unexpected arguments ()
scheme> (pair?)
Error: Unexpected arguments ()
scheme> (pair? 3)
#f
scheme> (pair? '(3 . 6))
#t
scheme> (pair? '(3 5))
#t
scheme> (number? '(3))
#f
scheme> (pair? '())
#f
scheme> (number? 3)
#t
scheme> (boolean? 3)
#f
scheme> (boolean? #f)
#t
scheme> (number? #t)
#f
scheme> (number?)
Error: Unexpected arguments ()
scheme> (boolean? 3 #f)
Error: Unexpected arguments (3 #f)
scheme> (null? '())
#t
scheme> (null? '('()))
#f
scheme> (null? '(3 5))
#f
scheme> (null?)
Error: Unexpected arguments ()
```

Problem 11. Extend the Runtime Library

Lastly, as we mentioned at the start, Scheme is a homoiconic language. This means code is data and data is code. For our very last function, we'll implement a primitive `eval` function, which takes as its argument a code datum and evaluates it to a value.

```
-- Primitive function `eval`
-- It evaluates the single argument as an expression
-- All you have to do is to check the number of arguments and
-- feed the single argument to the evaluator!
-- Examples:
--   (eval '(+ 1 2 3))  => 6
evalPrim :: [Val] -> EvalState Val
evalPrim = undefined
```

Testing

Aside from the provided testcases, you may want to manually enter the examples shown above, to observe that everything is working correctly (and for your benefit).

Finally, more cool stuff

This section is not graded, but you'll absolutely love it.

Now you have finished the MP! If you enjoyed implementing Scheme, you may enjoy these further challenges as well:

1. More parsing capabilities
    - The parser for the current REPL supports one expression per line. But you can extend it to accept multiple expressions in a single line.
    - Accepting file inputs will also be an important feature for a real programming language.
    - Comments begin with `;` and continue until the end of the line.
2. Memory model and side effects

    The subset of Scheme you've implemented so far is purely functional, but Scheme is not a purely functional language. Special form `set!` is used to modify a variable. Scheme also has a memory model, which has reference semantics. In particular, a closure that captures the environment is treated as a "heap object", which gets passed by reference.

    For example, Scheme lets us define a counter without using any global variables:

```
scheme> (define my-counter
          (let ((count 0))
            (lambda ()
              (set! count (+ count 1))
                count)))
scheme> (my-counter)
1
scheme> (my-counter)
2
scheme> (my-counter)
3
scheme> (define other-counter my-counter) ;; Assigning reference
scheme> (other-counter)
4
scheme> (other-counter)
5
scheme> (my-counter)
6
scheme> (other-counter)
7
```

Hint: You'll need to give `Env` an overhaul, simulate pointers (for which you might need the `IORef` monad), and properly handle global scoping and lexical scoping.

3. Higher-order functions

   Why not implement `map` and `reduce` in Scheme? Create your own functional library in Scheme, store the functions in a file, and load it every time you start the REPL.

4. Metacircular evaluator

   > Data is code. Code is data.

   With all the primitive functions that you implemented, did you know you can implement a Scheme evaluator in Scheme? Try to implement `eval` directly in Scheme!

   ```
   scheme> (eval '(+ 1 2 3))
   6
   scheme> (eval '(apply + '(1 2 3)))
   6
   ```

   Here's an introduction to the metacircular evaluator in one page:

   https://xuanji.appspot.com/isicp/4-1-metacircular.html

5. Learn everything else from SICP

Structure and Interpretation of Computer Programs is an excellent textbook for teaching the principles of programming.

Special thanks to Richard Wei for major contributions to this MP.