MP6 - Scheme

Logistics

- revision: 1.6 - Clarification about `=` and `eq?` edge case
- revision: 1.5 - Corrected sample output on several problems
- revision: 1.4 - Notes on `flattenList`, minimal REPL examples
- revision: 1.3 - Corrected various semantics, showed more environment state changes
- revision: 1.2 - Bugfix in `cond` rules; typesetting edited
- revision: 1.1 - Revised documentation for clarity
- release date: April 26, 2017
- due date: May 12, 2017 (end of day) - No later submissions!

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 an 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

As usual, you have to run `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 do it). 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

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

Available on Piazza:

- `mp6ref`: Executable Linux binary that works on EWS, as an illustrative reference for a model solution.

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`

```
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?

```
data Val = Symbol String
         | Boolean Bool
         | Number Int
         | DottedList [Val] Val
         | List [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 data type is `Val`, not `Exp`! Scheme, as well as other Lisps, is a homoiconic language, meaning that the text of the language and the values have the same structure as its AST.

> Code is data. Data is code.

Expressions from the programmer are now 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 mean, and provide some examples.

Kinds of values

1. `Symbol`

   Symbol is just a symbol. In previous MPs, a symbol expression, a.k.a. `SymExp`, was meant to be evaluated as a variable name bound to a value. In Scheme, this is no longer the case. Not only can you have values bound to a symbol, you can also use symbols themselves as a value.

2. `Boolean`

   Can be true `#t` or false `#f`.

3. `Number`

   An integer. We do not support floating point in this MP.

4. `List`

   A `List` (that is, a "proper" Scheme list; keep reading!) is a list of values terminated by an empty proper list. Scheme *theoretically* would define lists using *pairs* recursively, like this: a proper list would be a pair with a single item on the left and another proper list on the right. As a base case, there would be the special empty list `()`, which might be called "nil" or "null". However, we can get a speed advantage by simply using Haskell's native lists to implement our version of Scheme lists.

   In Scheme syntax, a proper list could be created like `(1 2 3)` or equivalently like `(1 . (2 . (3 . () )))` where the `.` is a pair separator.

5. `DottedList`

   Because Scheme theoretically uses pairs to make lists, we have a dilemma: what if the "list" isn't formed correctly as described above? A `DottedList` potentially represents an "improper" list of values: it can be ended by some *non-null* tail value (e.g. `(1 2 . 3)`). This is theoretically what you would get in Scheme if you kept nesting pairs on the right side of pairs, but where the rightmost nested item was *not* the empty list value, but some other value. In our implementation using Haskell, we make a type distinction between these "improper" dotted lists and the "proper" null-terminated lists, defining separate constructors: the `DottedList` constructor allows the trailing non-null value to be specified explicitly.

   This distinction allows us to utilize Haskell's native list data structure for better performance. This will also simplify our implementation of some other Scheme routines.

   However, because `DottedList` might be used to provide a null list as the trailing value, lists and dotted lists that are constructed differently can be equivalent by value. We offer you a useful helper function, `flattenList`, which flattens a `DottedList` to the simplest form; the flattening may simplify a dotted list to a proper list if possible, to a non-list singleton type, or to a simpler improper list that's still dotted. You will find it extremely useful when implementing primitive functions in the runtime, and you may want to use `flattenList` whenever you are trying to do Haskell pattern matching on a Scheme list. (That includes cases where you're passing a Scheme list to a Haskell function that has multiple definitions! Remember, in Haskell, that does pattern matching.)

For example, here are the results of flattening some proper and improper lists. When the left of the dot is blank, the right hand side is flattened recursively. The fourth example is an improper list, so it remains a dotted list after flattening.

Nested	Flattened
`( . ( . 3 ))`	`3`
`( . ())`	`()`
`(1 . ())`	`(1)`
`(1 . (2 . 3))`	`(1 2 . 3)`
`(1 . (2 . 3 . ()))`	`(1 2 3)`
`(1 . (2 3))`	`(1 2 3)`

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.

Example ASTs for Values

This table illustrates how some Scheme expressions are represented using our AST constructors of `Val` type. You can see how these are parsed for you in the `app/Scheme/Parse.hs` file. The quote expressions are explained later in the MP.

Scheme expression	Haskell AST representation
`1`	`Number 1`
`a`	`Symbol "a"`
`#t`	`Boolean True`
`#f`	`Boolean False`
`(define (f x) x)`	`List [Symbol "define", List [Symbol "f", Symbol "x"], Symbol "x"]`
`'1`	`List [Symbol "quote", Number 1]`
`` `,a ``	`List [Symbol "quasiquote", List [Symbol "unquote", Symbol "a"]]`

Diagnostic

You are given a fully defined `Diagnostic` type, cases of which are runtime errors thrown along evaluation. You are responsible for choosing the right `Diagnostic` to throw, in your 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: `Eval-`

`State` 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:

```
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:

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

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

You'll have to fill in parts of the REPL function from `Main.hs`, implementing cases for each of the possible results of evaluation. `repl :: Env -> IO ()`.

Our call to the function `runExcept` returns type `Either Diagnostic (Val, Env)`, where the `Either` type's `Right` pair represents the result value and new environment.

```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
        -- TODO:
        -- Insert line here: If error, print error
        -- 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
```

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

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

To start the REPL, run `stack ghci`, then call `main`.

```
$ stack ghci
[1 of 4] Compiling Scheme.Core (...)
[2 of 4] Compiling Scheme.Eval (...)
[3 of 4] Compiling Scheme.Parse (...)
[4 of 4] Compiling Scheme.Runtime (...)
[5 of 5] Compiling Main (...)
*Main Scheme.Core Scheme.Eval Scheme.Parse Scheme.Runtime> main
scheme>
```

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 evaluator and runtime, 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

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

Here, our notation may simply show a `Diagnostic` being returned in place of a value.

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

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

Problem 4. Special form `define` for variables

Now we want to allow the user to define variables. The variable definition form is `(define var exp)` (an s-expression). `var` must be a `Symbol`. The evaluator will evaluate `exp` and insert to the environment the value as a binding for the symbol. Use `modify` to mutate 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 nested `define`, but we don't check for that. For fun, you could try to use our version of the semantics to do something weird.)

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

We've already given you the ability to define functions. This 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 5b. Special form `lambda` for anonymous function expressions

You also need to implement a lambda-function form, `(lambda (params) body)`, which also 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.

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

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

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`

We should have some sort of if expression, because that's useful. Define the `(cond (c1 e1) ... (cn en))` form. If `c1` is not false, then `e1` is evaluated. If it's false the next condition should be tried.

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)`.

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

```
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 form `let*`

Define the `(let* ((x1 e1) ... (xn en)) body)` form.

Special form `let*` is like `let`, but evaluates $e_1, \ldots, e_n$ one by one, creating a location for each id as soon as the value is available. The ids are bound in the remaining binding expressions as well as the body, and the ids need not be distinct; later bindings shadow earlier bindings. (The notation $\sigma[x \mapsto v]$ shows an update to environment σ.)

$$\frac{[\![e_{\text{body}} \mid \sigma]\!] \Downarrow v}{[\![(\texttt{let*}\ ()\ e_{\text{body}}) \mid \sigma]\!] \Downarrow v}$$

$$\frac{[\![e_1 \mid \sigma]\!] \Downarrow v_1 \qquad [\![(\texttt{let*}\ ((x_2\ e_2) \cdots (x_n\ e_n))\ e_{\text{body}}) \mid \sigma[x_1 \mapsto v_1]\ ]\!] \Downarrow v}{[\![(\texttt{let*}\ ((x_1\ e_1) \cdots (x_n\ e_n))\ e_{\text{body}}) \mid \sigma]\!] \Downarrow v} \quad \text{where } n \geq 1$$

```
scheme> (let* ((x 5) (y (+ x 5))) (+ x y))
15
scheme> (define x 20)
scheme> (define y 30)
scheme> (let* ((x 11) (y x)) (- (* x y) 2))
119
scheme> x
20
scheme> y
30
```

Problem 9. Special forms `quote`, `quasiquote` and `unquote`

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
```

Problem 10. Special form `define-macro`

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 11. Application Form

If none of those forms were matched, then assume that the left-most part of the form is to be applied to the rest of the arguments. The format is:

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

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.

1. Application when `f` is a macro

    1. Save the environment (Hint: use `get` to get the environment from the state)
    2. Bind arguments (without evaluating them) to the parameters of the macro and insert them to the environment (Hint: use `modify` to mutate the state)
    3. Evaluate the macro body (i.e. expand the macro body)
    4. Restore the environment we saved in step 1
    5. Evaluate the expanded form (the result of step 3) and return it

2. Application in other cases

    Instead of handling other application form directly in `eval`, you are going to implement it as a separate function `apply :: Val -> [Val] -> EvalState Val`. It takes an applicable value (`Func`, `PrimFunc`) and applies it to a list of arguments. The list of arguments passed to `apply` are assumed to have been already evaluated.

    In `eval`, we evaluate the arguments and pass them to `apply`.

    In `apply`, we are going to handle three cases of first argument:

    - If it's a `Func`:
        1. Save the environment
        2. Insert bindings of the closure environment to the current environment
        3. Bind arguments to the parameters of the function and insert them to the environment
        4. Evaluate the function body
        5. Restore the environment we saved in step 1
        6. Return the result of step 4
    - If it's a `PrimFunc`, we directly apply the primitive function to the argument list
    - Otherwise, throw a diagnostic `CannotApply`.

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?

Runtime Library

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!

You need to initialize `runtime` with predefined primitive operators as well. This will make these operators available to users of your language. Some of them have already been filled in, however you will need to implement the rest.

You will not be able to do this right away! Even the cases that have been filled in use functions that are unimplemented. You'll need to implement a variety of lifting functions and primitive functions first.

```
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)
                     ]
```

We have provided the following translators to go between Scheme values and Haskell values. These can help when defining the various operator lifters. (Note that these take their single arguments in a Haskell list. See Problem 20 for more information.)

```
-- Primitive function `symbol?` predicate
isSymbol :: [Val] -> EvalState Val
isSymbol [Symbol _] = return $ Boolean True
isSymbol [_] = return $ Boolean False
isSymbol vv = throwError $ UnexpectedArgs vv

-- Primitive function `list?` predicate
```

```haskell
isList :: [Val] -> EvalState Val
isList [v] =
  return . Boolean $ case flattenList v of
    List _ -> True
    _ -> False
isList vv = throwError $ UnexpectedArgs vv
```

By the way, many of the provided stubs for the runtime problems use `const` to create a curried function that will simply discard any provided parameter and perform a stub action instead. For example,

```
Prelude> foo x y = "please implement this function"
Prelude> foo 1 2
"please implement this function"
Prelude> foo = const . const $ "please implement this function"
Prelude> foo 1 2
"please implement this function"
```

Operators

Problem 12. Variadic arithmetic operators (`+`, `-`, `*`, `/`): implement `liftIntVargOp`

Note: In this release of the assignment, this function has already been provided for you.

You need to implement `liftIntVargOp` that 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.

You should use `liftIntVargOp` to construct a `PrimFunc` for `+`, `-`, `*`, and `/`, and put it in `runtime`.

```
scheme> (+ 3 4 5)
12
scheme> (- 3 4 5)
-6
scheme> (-)
0
scheme> (+)
0
scheme> (* 3 5 9)
135
```

Problem 13. Variadic boolean operators (`and`, `or`): implement `liftBoolVargOp`

Unlike arithmetic operators which we have to apply pairwise to the list, variadic boolean operators, `and` and `or`, are provided by Haskell `Prelude`. You simply have to implement `liftBoolVargOp` to lift a function of type `[Bool] -> Bool` to a `PrimFunc`.

Boolean rule of thumb: In Scheme, everything except `#f` is considered true.

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

```
scheme> (and #t #t #t 'nil)
#t
scheme> (and #t #t #t #f)
#f
scheme> (and)
#t
scheme> (or)
#f
scheme> (or 't #t)
#t
scheme> (or 'to-be 'not-to-be)
#t
scheme> (or #f #f #f #f)
#f
scheme> (and 3 2 5)
#t
scheme> (and 3 2 5 #f)
#f
scheme> (or 3 2 5)
#t
```

Add boolean operators `and` and `or` to your `runtime` environment.

Problem 14. Binary comparison operators:

- `>` Integer greater than
- `<` Integer less than
- `>=` Integer greater than or equal
- `<=` Integer less than or 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`, i.e. `Boolean 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.

```
scheme> (< 3 4 5)
#t
```

```
scheme> (>= 3 4 2)
#f
scheme> (>=)
#t
scheme> (>= 7)
#t
```

Problem 15. List operators

These are the functions for composing and decomposing lists. For historical background on the naming of these functions, you might want to read this article:

https://en.wikipedia.org/wiki/CAR_and_CDR

- `car :: [Val] -> EvalState Val`

  Get the first element of the list or dotted list (*single argument*)

- `cdr :: [Val] -> EvalState Val`

  Get the rest of the list or dotted list (*single argument*)

- `cons :: [Val] -> EvalState Val`: Construct a `DottedList` from *2 arguments*

- `list :: [Val] -> EvalState Val`: Construct a `List` from as many arguments as given (hence it's *variadic*)

You must check the number of arguments, verify argument types, and throw appropriate `Diagnostics` on mismatch.

```
scheme> (car)
Error: Unexpected arguments ()
scheme> (cdr)
Error: Unexpected arguments ()
scheme> (car '(3 5))
3
scheme> (cdr '(3 5 6 7))
(5 6 7)
scheme> (cdr '(3 5 . 6))
(5 . 6)
scheme> (cdr '(3 5 . (6 . 7)))
(5 6 . 7)
scheme> (list (> 3 4) #t 15 #f (< 5 2 3 5) (cons 3 (cons 4 3)))
(#f #t 15 #f #f (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))
()
scheme> (cdr 'a)
Error: Unexpected arguments (a)
scheme> (cons 2 (cons 3 4))
(2 3 . 4)
scheme> (cons 2 (cons 3 (cons 4 #f)))
(2 3 4 . #f)
scheme> (cons 4 (cons 2 (cons 1 '('cs421 'is 'easy . ()))))
(4 2 1 (quote cs421) (quote is) (quote easy))
```

We want to put these unary operators in our `runtime`, which means we'll need to wrap them in a `PrimFunc`.

Problem 16. Unary boolean operator (`not`)

`not` is implemented for you. It's lifted from Haskell by `liftBoolUnaryOp`.

```
scheme> (not #t)
#f
scheme> (not #f)
#t
scheme> (not 'mattox)
#f
scheme> (not 'false)
#f
scheme> (not)
Error: Unexpected arguments ()
scheme> (not #f #t)
Error: Unexpected arguments (#f #t)
scheme> (not 'nil #t 3)
Error: Unexpected arguments (nil #t 3)
scheme> (not 3)
#f
```

Problem 17. Equality (`=`, `eq?`)

You will implement two variants of Scheme's equality:

- `=`

  Equality for numbers and booleans, throwing `TypeError` on type mismatch or unsupported types such as function

- `eq?`

  Equality for atom values, including numbers, booleans, and symbols, returning `#f` on type mismatch or unsupported types (not throwing diagnostics!)

Both `=` and `eq?` are variadic with default value `#t`. When the number of arguments is 0 or 1, these functions return `#t` *regardless* of the arguments' types or values.

```
scheme> (=)
#t
scheme> (= #f)
#t
scheme> (= #t #f #t)
#f
scheme> (= 1 1)
#t
scheme> (= 3 3 2)
#f
scheme> (= 'a)
#t
scheme> (= 'a 'a)
Error: Value a has unexpected type Symbol
scheme> (eq? 3 3 2)
#f
scheme> (eq? 'a 'a)
#t
```

Problem 18. Modulo (`modulo`)

Implement the `modulo` function (a binary operator taking two integers, and finding the remainder of one divided by the other, in accordance with Haskell's `mod` operator). Use `liftIntBinOp`.

```
scheme> (modulo)
Error: Unexpected arguments ()
scheme> (modulo 1)
Error: Unexpected arguments (1)
scheme> (modulo 1 5)
1
scheme> (modulo 5 1)
0
scheme> (modulo 5 2)
1
scheme> (modulo 6 (- 3))
0
```

```
scheme> (modulo (- 3) 1)
0
scheme> (modulo 7 (- 3))
-2
scheme> (modulo (- 7) 2)
1
scheme> (modulo 9 4)
1
```

Problem 19. Abs (`abs`)

Implement the `abs` function, which calculates the absolute value of the given integer. Use `liftIntUnaryOp`.

```
scheme> (abs)
Error: Unexpected arguments ()
scheme> (abs 1)
1
scheme> (abs 1 2)
Error: Unexpected arguments (1 2)
scheme> (abs (- 5))
5
```

Problem 20. 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`. We already showed you how to implement `symbol?` and `list?` earlier in this document! Check those out for hints.

- `symbol?` Checks whether the input is a `Symbol`.
- `list?` Checks whether a "flattened" version of the list is really a proper list (not a dotted list or something else).
- `pair?` Accepts either proper Scheme lists or dotted lists, nothing else. (Remember, in Scheme, both proper lists and dotted lists are technically types of pairs.) *However*, an empty list `()` is considered a "null pointer" just like an empty linked list, thus not a pair.
- `null?` Checks for an empty, proper list.
- `number?` Checks whether the input is a `Number`.

```
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 ()
```

Further steps

Problem 21. Extend the Runtime Library

Now that our evaluator is fully functional, we can implement the following functions that make use of the evaluator.

```
-- Primitive function `apply`
-- It applies a function to a list of parameters
-- Examples:
--   (apply + '(1 2 3))  => 6
--   (apply car '((1 2 3)))  => 1
--   (apply (lambda (x) (* 10 x)) '(300)) => 3000
applyPrim :: [Val] -> EvalState Val
applyPrim = undefined
```

```
-- 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
```

Now, add these functions to the runtime and test them.

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! We believe that you've had a lot of fun implementing Scheme. But the fun doesn't end here–what you have implemented is merely a subset of Scheme. Try to implement a few tasks below:

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.