Plan

1. ~~Overview of the Scheme programming language.~~
2. ~~Formalize the (almost-trivial) syntax of Scheme in Haskell.~~
3. ~~Develop an interpreter (evaluator) based on term-rewriting.~~
4. Develop an interpreter (evaluator) based on structural recursion (i.e. recursion with pattern matching).

This is similar to denotational semantics, which gives a meaning to each construct as a function of the meanings of its parts.

Likely-gratuitous Scheme example

```
;;; Using built-in integers
(define (factorial n)
  (if (eq? n 0)
        1
        (* n (factorial (- n 1)))))
```

```
;;; cons is the only provided way build data structures
(define (zip l0 l1)"
     (if (null? l0) "
        (list)"
         (if (null? l1)"
             (list)"
             (cons (cons (car l0) (car l1)) (zip (cdr l0) (cdr l1))))))"
```

Reminder: Scheme abstract syntax

```
data Exp
  = Atom String
  | List [Exp]
  | Number Integer
  | String String
  | Nil
  | Bool Bool
  deriving (Eq, Ord, Show)
```

Structure of rewrite-based intepreter

Scheme definitions

↓

Compile to rewrite rules

↓

Rewrite Scheme expression using the rules until no longer possible

The "semantics" approach

```
-- ignore typechecking issues for now
eval (Number n) = n
eval (String str) = str
eval (Nil) = nil
eval (Bool b) = b
eval (List (Atom f : es) = … (map eval es) …
--                           ^^^^^^^^^^^^^ argument values
eval (Atom str) = ?
```

What if `f` in the last line is

- `cons`
- `define`
- a defined function?

Values

Values are the result of evaluation. They no longer need to be in `Exp`.

```
data V
  = VNumber Int
  | VString String
  | VBool Bool
  | VNil
  | VCons V V
```

Suppose the Scheme program has a definition

```
(define (f x y) (+ x (* 2 y)))
```

Consider evaluating a call of `f`:

```
eval (List [Atom "f", List args) = … eval «(+ x (* 2 y))»? …
```

Environments

Two problems:

1. Where do variable values come from?
2. Where do function bodies come from?

Solution to both: *environments*.

```
-- An environment is a mapping x ↦ v ∈ V
newtype Env = Env {envMap :: Map String V}
```

Now:

```
eval :: Env -> Exp -> V
```

How can we store a function in an environment? What kind of value is a function?

Representing function values in V

```
(define (f x) (+ x 1))
(define (g y) (+ y 2))
(define (h z) (f (g z)) )
(h 17)
```

To evaluate `(h 17)` we need to evaluate `(f (g z))` with an environment where

- f ↦ ...
- g ↦ ...
- z ↦ 17

The first two "bindings" are from the point in the program where `h` is defined.
The third comes from the application of `h` to `17`

Closures

A value for `h` that we can store in an environment needs

- the parameter list `(z)`
- the bindings for what's available at `h`'s definition, i.e. `f` and `g`
- the body of the function

This is a *closure*. It contains everything needed to evaluate a call of the function.

```
data V
  = VNumber Int
  | VString String
  ⋮
  | VClosure Env [String] Exp
```

One remaining issue with closures

```
(define (m1 x) (- x 1))
(define (id y) (if (eq? y 0) 0 (+ 1 (id (m1 y)))))
```

`m1` closure `v`:

- env: empty
- vars: `x`
- body: `(- x 1)`

`id` closure:

- env: `m1 ↦ v`
- vars: `y`
- body: `(if (eq? y 0) 0 (+ 1 (id (m1 y))))`

The environment in a closure needs values for all the names in the function body.

So, it seems the environment built from the definitions needs

1. A binding `id ↦ v` where `v = VClosure env ["y"] (…)`
2. an environment `env` in the closure that itself has the binding `id ↦ v`

Actually, it doesn't need this circular closure.

We just need to make sure the `id` binding is there when we need it. See the code!