

psilo

This is the main program outline. If an argument is present on the command line then we execute the program in that file and halt. Otherwise we fire up a repl.

```
module Main where

import Parser
import Syntax
import Evaluator

import Control.Monad.Trans
import System.Console.Haskeline
import Control.Monad.Free
import Data.Monoid
import Data.Maybe
import Control.Monad (forM)

import System.Environment
import System.IO
import Text.Parsec
```

eval amounts to taking a list of parsed expressions and evaluating them in the context of a machine. The result is the state of the machine after it has been run.

```
eval :: Either ParseError [Expr ()] -> MStore -> IO [(Value, MStore)]
eval res store = do
  case res of
    Left err -> print err >> return [(VNil, store)]
    Right ex -> mapM execute (ex :: [Expr ()]) >>= return

  where execute v = do
    res <- (runMachineWithState store ev') . interpret $ v
    return res
    ev' = case (mGlobalEnv store) of
```

```

Nothing -> initialEnv
Just e -> MEnv e

```

The repl is nothing more than calling `eval` in an endless loop.

```

repl :: IO ()
repl = runInputT defaultSettings (loop initialStore) where
  loop store = do
    minput <- getInputLine "psilo> "
    case minput of
      Nothing -> outputStrLn "Goodbye."
      Just input -> do
        case input of
          ":state" -> liftIO (putStrLn . show $ store) >> loop store
          - -> do
              (val, store'):_ <- liftIO $ eval (parseToplevel input) store
              liftIO $ putStrLn . show $ val
              loop store'

```

If we are given a filename then we parse the code into an AST and make two passes. The first to collect all the definitions so that we can initialize the environment and store correctly. The second is to actually evaluate the program.

```

execFile :: String -> IO ()
execFile fname = do
  parsed <- liftIO $ parseFile fname
  case parsed of
    Left err -> print err >> return ()
    Right xs -> do
      defns <- forM xs $ \ expr -> do
        case expr of
          Free (ADefine sym val) -> do
            (_, store) <- runMachine . interpret $ expr
            return . Just $ store
          _ -> return Nothing
      sto <- return $ mconcat . catMaybes $ defns
      (sto':_) <- liftIO $ eval (Right (f xs)) sto
      return ()
  where f xs = filter (\x -> case x of
                                Free (ADefine _ _) -> False
                                - -> True) xs

main :: IO ()
main = do

```

```
args <- getArgs
case args of
  []      -> repl >> return ()
  [fname] -> execFile fname >> return ()
```