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 (parseFile, parseTopLevel)
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, forM_)
import Data.List (partition)

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

import Options.Applicative

data CmdLnOpts = CmdLnOpts {
      optRepl   :: Bool
    , optFile   :: String
    , optConLog :: Bool
} deriving Show

cmdLnOpts :: Parser CmdLnOpts
cmdLnOpts = CmdLnOpts
    <$> switch ( long "repl" <> short 'r' <> help "Initiate a REPL (default=TRUE)" )
    <*> strOption ( long "file" <> short 'f' <> help "Execute a file"
        <> value "" )
```

```
    <*> switch ( long "console-log" <> short 'l'
              <> help "Log debug output to the console (default=FALSE)"
    )
```

`evaluate` 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.

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

    where execute v = do
              ev <- return $ mGlobalEnv store
              res <- (runMachineWithState store (MEnv ev)) . eval $ v
              return res
```

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,log), store'):_ <- liftIO $ evaluate (parseTopLevel input) sto
                        liftIO $ putStrLn . show $ val
                        loop store'

execFile :: String -> Bool -> IO ()
execFile fname doLog = do
    parsed <- parseFile fname
    case parsed of
        Left err -> print err >> return ()
        Right xs -> do
            (defns, exprs) <- return $ partition isDefn xs
```

```
            initState <- initializeState defns initialStore
            final <- evaluate (Right exprs) initState
            return ()
    where isDefn (Free (ADefine _ _)) = True
          isDefn _                    = False
          initializeState [] sto = return sto
          initializeState ds sto = do
              (_,r) <- (flip runMachine) doLog $ forM_ ds eval
              return r

main :: IO ()
main = execParser opts >>= start

start :: CmdLnOpts -> IO ()
start os = if doRepl then repl else case doFile of
    "" -> return ()
    fname -> execFile fname conLog
    where
        doRepl = optRepl os
        doFile = optFile os
        conLog = optConLog os

opts :: ParserInfo CmdLnOpts
opts = info (cmdLnOpts <**> helper)
    ( fullDesc <> progDesc "Run psilo programs" <> header "psilo" )
```