The psilo Virtual Machine

The virtual machine consists of two things:

- a statically scoped mapping from symbols to store locations (the "environment"); and
- a persistent mapping from locations to values (the "store").

Thus a `Machine` is a monad composed of `ReaderT` and `StateT` monad transformers.

The Reader monad permits function-local overwriting of the contained state which is automatically rolled back – precisely the behavior we want out of our lexical environment.

The State monad, on the other hand, is persistent until the end of the machine's execution and thus handles dynamic scope and state.

Please note that this is simply a reference implementation of the virtual machine to start playing with psilo's grammar and other features; by no means is this intended to be efficient, production-quality software.

Imports and language extensions

```
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveTraversable #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE OverlappingInstances #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE ExistentialQuantification #-}

module Evaluator where

import Control.Monad.Free
import Prelude hiding (log,lookup)
```

```haskell
import Control.Monad
import Control.Monad.State
import Control.Monad.Free
import Control.Monad.Trans
import Control.Monad.Reader
import qualified Data.Map.Strict as Map
import qualified Data.IntMap.Strict as IntMap
import Data.Foldable (Foldable, fold)
import Data.Traversable (Traversable, sequence)
import Data.List (intersperse, nub)

import Parser
import Syntax
```

The Machine

Borrowing (stealing?) from Krishnamurthi's inimitable Programming Languages: Application and Interpretation the environment does not map symbols to values but to *locations* in the store. The store, then, maps location to values.

```haskell
type Location = Int
data Value = forall a . Show a => VClos { vSym  :: [Symbol]
                                        , vBody :: (Expr a)
                                        , vEnv  :: [(Symbol, Value)]
                                        }
           | VSym Symbol
           | VNum  { unNum :: Integer }
           | VBool { unBool :: Bool }
           | VList [Value]
           | VDefine Symbol
           | VNil

instance Eq Value where
    (VNum a)  == (VNum b)  = a == b
    (VBool a) == (VBool b) = a == b
    (VSym a)  == (VSym b)  = a == b
    _         == _         = False

instance Show Value where
    show (VSym s)   = "'" ++ s
    show (VNum n)   = show n
    show (VBool b)  = if b then "#t" else "#f"
    show (VNil)     = "(nil)"
    show (VClos _ _ _)  = "<function>"
    show (VList xs) = concat $ map show xs
```

```
    show (VDefine _)= "<definition>"

type Environment = Map.Map Symbol Int
type Store       = IntMap.IntMap Value

emptyEnv = Map.empty
emptyStore = IntMap.empty
```

The store must also keep track of how many locations it has handed out. As the `StateT` monad can only hold one value as state, I wrap a `Store` and an `Int` together in one data type.

```
data MStore = MStore { mStore :: Store
                     , mLoc     :: Int
                     , mFinalEnv :: Maybe Environment
                     }
    deriving Show
```

I do the same thing with `Environment` defensively in case I need to store more data in the `ReaderT` in the future.

```
data MEnv = MEnv { mEnv :: Environment }
    deriving Show
```

Behold: the `Machine` monad, a stack of monad transformers.

```
newtype Machine a = M { runM :: ReaderT MEnv (StateT MStore IO) a }
    deriving (Monad, MonadIO, MonadState MStore, MonadReader MEnv)

initialStore :: MStore
initialStore = MStore { mStore = emptyStore
                      , mLoc     = 1
                      , mFinalEnv = Nothing
                      }

initialEnv :: MEnv
initialEnv = MEnv { mEnv = emptyEnv }
```

With the above default initial states for the environment and the store, I'm ready to define the mapping from a `Machine` to `IO`, which is essentially just calling the various monad transformer `run` functions in succession.

```
runMachineWithState :: MStore -> MEnv -> Machine a -> IO (a, MStore)
runMachineWithState st ev k = runStateT (runReaderT (runM k) ev) st where
```

```haskell
runMachine :: Machine a -> IO (a, MStore)
runMachine k = runMachineWithState initialStore initialEnv k
```

Now all that is left is a means of building a `Machine` from psilo code.

Interpreting psilo

Now that we have a static environment, a dynamic store, and a machine which holds the two, we can set ourselves to interpreting psilo.

As stated elsewhere, executing psilo programs is the act of

1. transforming `Expr` values to `Machine` values and
2. unwinding `Machine` values.

Some common operations have been factored out into helper functions, viz:

```haskell
fresh :: Machine Location
fresh = do
    state <- get
    loc   <- return $ mLoc state
    put $ state { mLoc = (loc + 1) }
    return loc

fetch :: Location -> Machine (Maybe Value)
fetch loc = do
    state <- get
    sto   <- return $ mStore state
    val   <- return $ IntMap.lookup loc sto
    return val

lookup :: Symbol -> Machine Value
lookup sym = do
    MEnv env <- ask
    loc      <- return $ Map.lookup sym env
    case loc of
        Nothing -> do
            return VNil
        Just loc' -> do
            mv       <- fetch loc'
            case mv of
                Nothing -> do
                    liftIO $ putStrLn $ "loc = " ++ show loc
                    return VNil
                Just  v -> return v
```

4

```haskell
store :: Location -> Value -> Machine ()
store loc val = do
    state <- get
    sto   <- return $ mStore state
    sto'  <- return $ IntMap.insert loc val sto
    put $ state { mStore = sto' }


delete :: Location -> Machine ()
delete loc = do
    state <- get
    sto   <- return $ mStore state
    sto'  <- return $ IntMap.delete loc sto
    put $ state { mStore = sto' }


bind :: Symbol -> Value -> Machine a -> Machine a
bind sym val next = do
    newLoc <- fresh
    store newLoc val
    state <- get
    maybeGlobalEnv <- return $ mFinalEnv state
    globalEnv <- case maybeGlobalEnv of
        Nothing -> return $ Map.fromList []
        Just e  -> return e
    let globalEnv' = Map.union (Map.fromList [(sym,newLoc)]) globalEnv
    put $ state { mFinalEnv = Just globalEnv' }
    local (\_ -> MEnv globalEnv') next
```

The function `interpret` handles the first part. You can even tell by its type: `Expr a -> Machine Value`.

```haskell
interpret :: Show a => Expr a -> Machine Value
```

`Expr` is defined as `type Expr = Free AST`. Since `Expr` is a `Free` monad, as with `Op`, we handle the base case of being handed a `Pure` value. In this case, we return `NilV`.

```haskell
interpret (Pure _) = return VNil
```

Numbers and Booleans are easy enough to deal with:

```haskell
interpret (Free (AInteger n)) = return $ VNum n
interpret (Free (ABoolean b)) = return $ VBool b
```

Symbols are slightly more interesting. We must lookup the location of the symbol's value in the environment, and then its value using the location.

```
interpret (Free (ASymbol s)) = lookup s >>= return
```

Lists are handled by iterating over the list of `Expr` values and constructing a list of `Values`, which we wrap in `VList`.

```
interpret (Free (AList xs)) = do
    vals <- forM xs $ \x -> do
        x' <- return x
        let v = interpret x'
        v
    return $ VList vals
```

Function abstraction amounts to creating a closure; that is to say, an environment and a body expression. The environment is essentially a new frame that will be temporarily prepended to the main environment when the body is evaluated.

```
interpret (Free (ALambda args body)) = do
    vars <- variables body
    vars' <- forM vars $ \var -> do
        val <- lookup var
        return (var, val)
    return $ VClos args body vars'
```

Function application works by first checking to see if the operator is a built-in. If not, we must do the following:

1. Lookup the closure in the machine's environment.
2. Augment the current environment with that of the closure.
3. Evaluate the body of the closure.
4. Roll back the changes to the environment.
5. Return the value.

```
interpret (Free (AApply op args)) = do
    VList args' <- interpret args
    res   <- builtin op args'
    oldState <- get
    case res of
        Just v -> return v
        Nothing -> do
            (VClos syms body env) <- interpret op
```

```
closedEnv <- forM env $ \(s, val) -> do
    loc <- fresh
    store loc val
    return (s, loc)
closedEnv' <- return $ Map.fromList closedEnv
argEnv <- forM (zip syms args') $ \(sym, av) -> do
    loc <- fresh
    store loc av
    return (sym, loc)
argEnv' <- return $ Map.fromList argEnv
newEnv <- return $ Map.union argEnv' closedEnv'
retVal <- local (\(MEnv e) -> MEnv (Map.union newEnv e)) $
    interpret body
put oldState
return retVal
```

Definitions are handled differently than other expressions because, really, they're
not expressions. You can't meaningfully compose definitions. They are simply a
guarded mechanism for the programmer to modify the global environment.

```
interpret (Free (ADefine sym val)) = do
    val' <- interpret val
    bind sym val' $ return $ VDefine sym
```

To construct an appropriate environment, we must find all the free variables in
the body expression.

```
variables :: Expr a -> Machine [Symbol]
variables (Free (ASymbol s)) = return [s]

variables (Free (AList xs)) = do
    listOfVarLists <- mapM variables xs
    vars           <- return $ nub $ concat listOfVarLists
    return vars

variables (Free (AApply op args)) = do
    varList <- variables args
    return varList

variables _   = return []
```

Built-in operators

Some symbols denote built-in operators (mostly involving arithmetic). The
following function attempts to evaluate a built-in, returning (maybe) a `Machine`
`Value`.

```haskell
--builtin :: Expr a -> [Value] -> Machine (Maybe Value)
builtin (Free (ASymbol sym)) args
    | sym == "+"    = numOp sum args
    | sym == "*"    = numOp product args
    | sym == "-"    = numBinOp ((-)) args
    | sym == "/"    = numBinOp div   args
    | sym == "=?"   = return $ Just . VBool $ (args !! 0) == (args !! 1)
    | sym == "if"   = return . Just $ boolIf args
    | sym == "and"  = return $ Just . VBool $ and (map unBool args)
    | sym == "or"   = return $ Just . VBool $ or  (map unBool args)
    | sym == "not"  = return $ Just . VBool $ not (unBool . head $ args)
    | otherwise     = return  Nothing
    where numBinOp op xs = let (VNum l) = xs !! 0
                               (VNum r) = xs !! 1
                           in  return $ Just . VNum $ op l r
          numOp    op xs = return $ Just . VNum $ op (map unNum args)
          boolIf xs = if (unBool (xs !! 0)) then (xs !! 1) else (xs !! 2)


builtin (Free (AInteger n)) _ = return $ Just . VNum $ n
builtin (Free (ABoolean b)) _ = return $ Just . VBool $ b
builtin _ _   = return Nothing
```