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

```haskell
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE OverlappingInstances #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE ExistentialQuantification #-}

module Evaluator where

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

```haskell
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 Data.Monoid

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 Ord Value where
    (VNum a) <= (VNum 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)"
```

```haskell
    show (VClos _ _ e)  = "<function> { " ++ (show e) ++ " } "
    show (VList xs) = concat $ map show xs
    show (VDefine _)= "<definition>"

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

emptyEnv = Map.empty
{-# INLINE emptyEnv #-}
emptyStore = IntMap.empty
{-# INLINE emptyStore #-}
```

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.

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

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

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

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

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

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

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

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

```
runMachineWithStore st k = runStateT (runReaderT (runM k) initialEnv) st

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

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.

I define `Monoid` instances for my `Environment` and `Store` types for the benefit of the interpreter. For each `=` in the source code, I evaluate the statement to obtain a machine state containing the function definition. Once completed I merge the machine states together using `mconcat` fromsecond `Data.Monoid`.

```
shiftKeysBy n m = IntMap.mapKeys (+n) m
shiftValsBy n m = Map.map (+n) m

instance Monoid (Map.Map Symbol Int) where
    mempty = emptyEnv
    a `mappend` b = Map.union a (shiftValsBy (Map.size a) b)

instance Monoid (IntMap.IntMap Value) where
    mempty = emptyStore
    a `mappend` b = IntMap.union a (shiftKeysBy (IntMap.size a) b)

instance Monoid MStore where
    mempty = initialStore
    a `mappend` b = MStore {
        mStore = (mStore a) `mappend` (mStore b),
        mLoc   = (mLoc a) + (mLoc b),
        mGlobalEnv = (mGlobalEnv a) `mappend` (mGlobalEnv b)
    }
```

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:

```
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 localEnv <- ask
    state <- get
    case (mGlobalEnv state) of
        Nothing -> lookup' sym localEnv
        Just  e -> lookup' sym (Map.union localEnv e)
    where
        lookup' sym env = do
            maybeLoc <- return $ Map.lookup sym env
            case maybeLoc of
                Nothing -> return VNil
                Just loc -> do
                    maybeVal <- fetch loc
                    case maybeVal of
                        Nothing -> return VNil
                        Just  v -> return v

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' }

bind :: Symbol -> Value -> Machine a -> Machine a
bind sym val next = do
    newLoc <- fresh
```

```
    store newLoc val
    state <- get
    maybeGlobalEnv <- return $ mGlobalEnv state
    globalEnv <- case maybeGlobalEnv of
        Nothing -> return $ Map.fromList []
        Just e  -> return e
    let globalEnv' = (Map.fromList [(sym,newLoc)]) <> globalEnv
    put $ state { mGlobalEnv = Just globalEnv' }
    local (\_ -> MEnv globalEnv') next
```

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

```
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`.

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

Numbers and Booleans are easy enough to deal with:

```
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)) = do
    val <- lookup s
    lookup s >>= return
```

Lists are handled by iterating over the list of `Expr` values and constructing a list of `Value`s, 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
    Free (AList args') <- return args
    Free op' <- return op
    isBuiltin <- builtin op args'
    case isBuiltin of
        Just k ->  return k
        Nothing -> do
            (interpret op) >>= handle
    where
      handle (VClos syms body closedEnv) = do
          oldState <- get
          closedEnv' <- forM closedEnv $ \(s, val) -> do
              loc <- fresh
              store loc val
              return (s, loc)
          closure <- return $ Map.fromList closedEnv'
          VList args' <- interpret args
          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' closure
          retVal <- local (\(MEnv e) -> MEnv (Map.union newEnv e)) $
```

```
            interpret body
        put oldState
        return retVal
    handle (VSym sym) = lookup sym >>= handle
    handle _          = return VNil
```

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
    liftIO . putStrLn $ "Defining <" ++ (show sym)
                     ++ "> = " ++ (show 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 (Free (ALambda syms body)) = do
    bodyVars <- variables body
    return $ bodyVars

variables _   = return []
```

Built-in operators

There are a few operators which are not core to the language *per se* but which either we tend to take for granted as being in a language or which would be unreasonably difficult to actually write in psilo.

For example: right now, by default all function arguments are evaluated before being passed to their respective functions. However, `if` must only evaluate one or the other of its operands, otherwise really bad things could happen if you depended on it for recursion.

NB: this could be the foundation of a macro system now that I think about it, if this table could be extended by psilo code …

```
builtin (Free (ASymbol sym)) xs
    | sym == "+"   = numsOp sum xs
    | sym == "*"   = numsOp product xs
    | sym == "-"   = numBinOp ((-)) xs
    | sym == "/"   = numBinOp (div) xs
    | sym == "=?"  = boolEq xs
    | sym == "if"  = boolIf xs
    | sym == "<"   = boolBinOp (<) xs
    | sym == ">"   = boolBinOp (>) xs
    | sym == "<="  = boolBinOp (<=) xs
    | sym == ">="  = boolBinOp (>=) xs
    | sym == "and" = boolBinOp (\(VBool x) (VBool y) -> and [x,y]) xs
    | sym == "or"  = boolBinOp (\(VBool x) (VBool y) -> or  [x,y]) xs
    | sym == "not" = boolOp (\(VBool x) -> not x) xs
    | sym == "print" = do
        xs' <- mapM interpret xs
        liftIO $ forM_ xs' (putStrLn . show)
        return . Just $ VNil
    | otherwise   = return Nothing
   where
       numsOp op xs = do
           xs' <- mapM interpret xs
           return . Just . VNum $ op (map unNum xs')
       numBinOp op (l:r:_) = do
           VNum l' <- interpret l
           VNum r' <- interpret r
           return . Just . VNum $ op l' r'
       boolEq (l:r:_) = do
           l' <- interpret l
           r' <- interpret r
           return . Just . VBool $ l' == r'
       boolIf (c:t:e:_) = do
           VBool cond <- interpret c
           if cond
               then interpret t >>= return . Just
               else interpret e >>= return . Just
       boolOp op (x:_) = do
           x' <- interpret x
           return . Just . VBool $ op x'
       boolBinOp op (l:r:_) = do
```

```
          l' <- interpret l
          r' <- interpret r
          return . Just . VBool $ op l' r'
builtin (Free (AInteger n)) _ = return . Just . VNum $ n
builtin (Free (ABoolean b)) _ = return . Just . VBool $ b

builtin _ _ = return Nothing
```

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