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.

Imports and language extensions

```haskell
{-# 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)
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)

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.

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

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 _ _ e)   = "<function> with Environment: " ++ (show e)
    show (VList xs) = concat $ map show xs

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.

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

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
```

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)

initialStore :: MStore
initialStore = MStore { mStore = emptyStore
                      , mLoc   = 1
                      }

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.

```haskell
runMachineWithStore :: Machine a -> MStore -> IO (a, MStore)
runMachineWithStore k st = runStateT (runReaderT (runM k) initialEnv) st

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

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

The operation language

Executing a psilo program on this machine amounts to:

1. Unwinding `Expr` values and concurrently
2. Building the corresponding `Machine` values.

To aid in this second step I define an intermediate operation language, `Op`, which will encapsulate some common machine-oriented tasks. This should simplify the proper interpreter function.

`Op` allows for manipulation of the environment and store, and also provides fresh store locations on demand.

```
data OpF k
    = Bind   Symbol Location k
    | Lookup Symbol (Location -> k)
    | Store  Location Value k
    | Fetch  Location (Value -> k)
    | Delete Location k
    | Fresh  (Location -> k)
    deriving Functor

type Op = Free OpF
```

Nevermind the `Free` constructor for now.

The utility of the following convenience functions will be clearer when we get to the interpreter. A crude explanation is that these are the "commands" we will write `Op` programs in.

```
bind :: Symbol -> Location -> Op ()
bind s l = liftF $ Bind s l ()

lookup :: Symbol -> Op Location
lookup s = liftF $ Lookup s id

store :: Location -> Value -> Op ()
store l v = liftF $ Store l v ()

fetch :: Location -> Op Value
fetch l = liftF $ Fetch l id

delete :: Location -> Op ()
delete l = liftF $ Delete l ()

fresh :: Op Location
fresh = liftF $ Fresh id

lookupVar :: Symbol -> MEnv -> Location
lookupVar sym (MEnv env) = env Map.! sym

bindVar :: Symbol -> Location -> MEnv -> MEnv
bindVar sym loc (MEnv env) = MEnv $ Map.insert sym loc env
```

The `Op` interpreter and the `Free` monad

```
runOp :: Op a -> Machine a
```

For each branch of our `OpF` data type definition, we have a case for the `Op` interpreter to handle.

`Op` and `OpF` are slightly different: the former is the latter transformed by the `Free` monad type constructor. `Free` is exactly that: you give it a functor value and get a monad for "free"; ie, you get a data type which implements `>>=` and `return`.

However, `Free` monads all get the same generic implementations of `>>=` and `bind`. All the semantics of your data type must be specified in a *run* function of some kind which breaks down these aggregate values to build some result. `runOp` is such a function for `Op`.

`Free` creates types which have a base case for storing "Pure" values (in Haskell, the `return` function may also be called `pure` because it wraps a *pure* value in a monadic context). `Free` values all have a continuation argument (which I call `next`, conventionally), whereas `Pure` values do not: they are leaves of a syntax tree.

```
runOp (Pure v) = return v
```

All the other value constructors are wrapped in `Free`.

`binding` is the act of associating a symbol with a location in memory.

```
runOp (Free (Bind sym loc next)) = do
    env <- ask
    local (bindVar sym loc) $ runOp next
```

The inverse operation is called a `lookup` in our parlance:

```
runOp (Free (Lookup sym next)) = do
    loc <- asks (lookupVar sym)
    runOp $ next loc
```

If we have a location (perhaps by calling `fresh`) to store a value in - and a value - we can associate them by `get`ting the state, picking out the `Store`, and inserting our location and value in the map. Then we `put` the state back in and continue on our merry way.

```
runOp (Free (Store loc val next)) = do
    state  <- get
    sto    <- return $ mStore state
    sto'   <- return $ IntMap.insert loc val sto
    put $ state { mStore = sto' }
    runOp next
```

By this point should be able to figure out what `fetch` does.

```
runOp (Free (Fetch loc next)) = do
    state  <- get
    sto    <- return $ mStore state
    val    <- return $ sto IntMap.! loc
    runOp $ next val

runOp (Free (Delete loc next)) = do
    state <- get
    sto   <- return $ mStore state
    sto'  <- return $ IntMap.delete loc sto
    put $ state { mStore = sto' }
    runOp next
```

`fresh` gets the state, extracts the location, increments it, puts it back in, and returns the original.

```
runOp (Free (Fresh next)) = do
    state <- get
    loc   <- return $ mLoc state
    put $ state { mLoc = (loc + 1) }
    runOp $ next loc
```

To illustrate what `Op` code looks like have a look at `opTest`:

```
opTest :: Machine ()
opTest = runOp $ do
    loc1 <- fresh
    bind "huh" loc1
    store loc1 $ VNum 5
    (VNum val) <- lookup "huh" >>= fetch
    bind "huh" 0
    store 0 $ VNum (val * 2)
    return ()
```

Interpreting psilo

Now that we have a static environment and a dynamic store, a machine which holds them, and a low-level operation language to control the machine, we can now 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.

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 v) = return VNil
```

The rest of the interpreter is remarkably simple. Each case corresponds to a branch in our `AST` definition.

```
interpret (Free (ABoolean b)) = return $ VBool b

interpret (Free (AInteger n)) = return $ VNum n
```

Symbols may have prefixes or suffixes (well, eventually) which modify the semantic value of the symbol but not the actual raw value. For example, a `:&` suffix tells the compiler that the symbol is a shared reference and may be safely ignored.

While this will be more nuanced or sophisticated in the future, in this evaluator at least we may safely ignore all suffixes.

```
interpret (Free (ASymbol  s)) = do
    val <- runOp $ lookup s >>= fetch
    return val

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

The below code for lambdas, while technically correct, has a huge problem: it copies its *entire* environment. A much smarter trick would be to only copy that which is actually used.

Also, function application is currently very stupid. It is intended that functions will take one argument, a list containing the actual values to be processed. The process is simple:

1. If the operand list is sufficiently long, zip it with the list of symbols in the function.
2. Create an `Environment` out of this zipped list.
3. Form a union between this new environment and the current, favoring the new one.

```
interpret (Free (ALambda args body)) = do
    (MEnv currentEnv) <- ask
    return $ VClos args body currentEnv
```

During application, if we are given a symbol for an operator, check to see if it is a built-in operator and, if applicable, simply return the resulting `Value`.

```
interpret (Free (AApply fun args)) = do
    (VList argVals)       <- interpret args
    case builtin fun argVals of
        Just mv    -> return mv
        Nothing    -> do
            (VClos syms body env) <- interpret fun
            locations <- forM argVals $ \av -> do
                newLoc <- runOp $ fresh
                runOp $ store newLoc av
                return newLoc
            let env' = Map.fromList $ zip syms locations
            newFrame <- return $ Map.union env' env
            oldEnv   <- ask
            retVal   <- local (\(MEnv e) -> MEnv (Map.union newFrame e)) $
                            interpret body
            -- clean up the store
            runOp $ forM_ locations $ \loc -> delete loc
            return retVal
```

This interpreter is flexible and powerful because we built up the appropriate abstractions.

Built-in operators

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

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

builtin _ _    = Nothing
```