Homework 4 Clarifications

- Use the given types.

- Upon encountering an error, the interpreter should stop that computation thread and return the error in the current state, with the current output.

  test $ (Amb (Out (Con 1) &: i) (Out (Con 2) &: Skip)) :$
  (Out (Con 3) & TT)

  will evaluate to:

  (Success: True | Output: 1; 3; | MyState {step = 1, refs = []})
  (Error: expected function, found () |
  Output: 2; 3; | MyState {step = 0, refs = []})

Note: interpreter evaluates both parts of application before checking for error.
• Let can also be desugared as:

\[
\text{Let } [x_0 := e_0, \ldots, x_n := e_n] e \equiv (x_0 : \ldots (x_n : e) : e_n \ldots) : e_0
\]

which has the same operational semantics as the desugaring given in the homework:

\[
(x_0 : \ldots x_n : e) : e_0 : \ldots e_n
\]

Similarly for LetRec.
The State Monad

newtype State s a = State (s -> (a,s))

instance Monad (State s) where
  return x = State $ \s -> (x,s)
  State x >>= f = State $ \s0 ->
    let (v1,s1) = x s0
        State x2 = f v1
    in x2 s1

get :: State s s
get = State $ \s -> (s,s)

put :: s -> State s ()
put s = State $ \_ -> (() ,s)
newtype Ref = Ref Int deriving Eq

data Value = ...

type St = [(Ref, Value)]

newRef :: Value -> State St Ref
newRef v = do
    st <- get
    let ref = generateNewRef st
    put $ (ref, v) : st
    return ref

getRef :: Ref -> State St Value
getRef ref = do
    st <- get
    let Just v = lookup ref st
    return v

setRef :: Ref -> Value -> State St ()
setRef ref v = do
    st <- get
    put $ (ref, v) : st
    return ()
The Reader Monad

newtype Reader r a = Reader (r -> a)

instance Monad (Reader r) where
    return x = Reader $ \_ -> x
    Reader x >>= f = Reader $ \r ->
        let Reader x' = f $ x r
        in x' r

ask :: Reader r r
ask = Reader $ \r -> r

local :: (r -> r) -> Reader r a -> Reader r a
local f (Reader x) = Reader $ \r -> x $ f r
newtype Writer w a = Writer (a,w)

instance Monad (Writer w) where
  return x = Writer (x, ???)
  Writer (x,w) >>= f =
    let Writer (v,w’) = f x
    in Writer (v, ???)

tell :: w -> Writer w ()
tell w = Writer ((), w)

We need some general way of denoting no output and combining output.
The Monoid Class

class Monoid a where
    mempty :: a
    mappend :: a -> a -> a

Of course, mappend should be associative and have mempty as a left and right unit.

Some common instances:

instance Monoid [a] where
    mempty = []       -- empty list
    mappend = (++)    -- list append

instance Monoid (a -> a) where
    mempty = id      -- identity function
    mappend = (.)    -- function composition
The Writer Monad

newtype Writer w a = Writer (a, w)

instance Monoid w => Monad (Writer w) where
  return x = Writer (x, mempty)
  Writer (x, w) >>= f =
    let Writer (v, w') = f x
    in Writer (v, mappend w w')

tell :: w -> Writer w ()
tell w = Writer ((), w)
The Real World

- We can use the State monad to model interaction with the real world.

- Create an abstract type, World, to be the state of the world, e.g. input stream, output stream, file system, etc...

- Operations involving the world will have type: State World a

- Provide interface functions to operating system primitives, e.g. reading and writing to streams.

- Note destructive updates of World are safe since monadic combinators guarantee single threaded state use.
The IO Monad

In Haskell, input/output operations are done in a special monad called IO.

putChar :: Char -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO () -- adds a newline
print :: Show a => a -> IO ()

getChar :: IO Char
getLine :: IO String
getContents :: IO String
interact :: (String -> String) -> IO ()
readIO :: Read a => String -> IO a
readLn :: Read a => IO a

The Read class contains functions to convert a string to a Haskell value.

IO is an abstract type to prevent the user from directly manipulating the world.
Native References

The single threadedness of monadic state also allows for safely implementing references with destructive updates.

\[
\begin{align*}
\text{newIORef} & : a \to \text{IO (IORef } a) \\
\text{readIORef} & : \text{IORef } a \to \text{IO } a \\
\text{writeIORef} & : \text{IORef } a \to a \to \text{IO } ()
\end{align*}
\]

IORef is an abstract type constructor.

Note this allows creation of a reference to an arbitrary type. This mechanism cannot be encoded in Haskell.

GHC also provides the \texttt{ST} monad for just state, with references, but not I/O.