□ Monad Transformer

```
class MonadT t where
    lift :: Monad m => m a -> t m a
```

Additional requirements:

- `(t m)` is a monad for any monad `m`.
- `lift . return`$_{\texttt{m}}$ $\equiv$ `return`$_{\texttt{t m}}$
- `lift (x >>=`$_{\texttt{m}}$ `k)` $\equiv$ `(lift x) >>=`$_{\texttt{t m}}$ `(lift . k)`

□ State Monad Transformer

```
class MonadT t where
    lift :: Monad m => m a -> t m a

class Monad m => StateM s m where
    update :: (s -> s) -> m s

newtype StateT s m a = StateT {runStateT :: s -> m (a, s)}

instance Monad m => Monad (StateT s m) where
    return x = StateT $ \s -> return (x, s)
    x >>= f = StateT $ \s0 -> runStateT x s0 >>= (\(v,s1) ->
                              runStateT (f v) s1)

instance MonadT (StateT s) where
    lift x = StateT $ \s0 -> x >>= (\v -> return (v, s0))

instance Monad m => StateM s (StateT s m) where
    update f = StateT $ \s -> return (s, f s)

instance (StateM s m, MonadT t, Monad (t m)) => StateM s (t m) where
    update = lift . update
```

□ Error Monad Transformer

```
data Error a = Ok a | Err String

class Monad m => ErrorM m where
    err :: String -> m a

newtype ErrorT m a = ErrorT {runErrorT :: m (Error a)}

instance Monad m => Monad (ErrorT m) where
    return x = ErrorT $ return $ Ok x
    x >>= f = ErrorT $ runErrorT x >>= (\v1 ->
        case v1 of Ok v2 -> runErrorT $ f v2
                   Err s -> return $ Err s
      )

instance MonadT ErrorT where
    lift x = ErrorT $ x >>= (\v -> return $ Ok v)

instance Monad m => ErrorM (ErrorT m) where
    err s = ErrorT $ return $ Err s

instance (ErrorM m, MonadT t, Monad (t m)) => ErrorM (t m) where
    err = lift . err
```

□ Desugaring Subtleties

- `Let [x1 :=: e1, x2 :=: e2] e` $\equiv$ `(x1 :\ x2 :\ e) :$: e1 :$: e2`
  implies
  `Let [x1 :=: e1, x2 :=: e2] e` $\not\equiv$ `Let [x1 :=: e1] (Let [x2 :=: e2] e)`
  because of possible variable capture.

  `Let [x1 :=: e1, x2 :=: e2] e` $\equiv$ `(x1 :\ (x2 :\ e) :$: e2) :$: e1`
  implies
  `Let [x1 :=: e1, x2 :=: e2] e` $\equiv$ `Let [x1 :=: e1] (Let [x2 :=: e2] e)`

- `While e1 e2` $\equiv$

  ```
  LetRec [w :=: "_" :\ If e1 (e2 :& w :$: Skip) Skip] (Var w :$: Skip)
  ```

  where `w` not free in `e1` nor `e2`

  `While e1 e2` $\equiv$ `If e1 (e2 :& While e1 e2) Skip`
  also works, but not really desugaring.

□ Homework 5

New types for intepreter from homework 4.

```
interp :: Term -> M a
interp (Var x) = do
    env <- ask
    getVar env x
interp (x :\ e) = do
    env <- ask
    return $ Fun $ \v -> local (\_ -> (x,v):env) $ interp e
interp (e1 :$: e2) = do
    v1 <- interp e1
    v2 <- interp e2
    apply v1 v2
  :
  :

test :: Term -> ((Either String Value, String), MyState)
test e = run initSt [] $ interp e
```

Note no non-determinism, so `Term` has no `Amb` constructor; also `interp` now uses the built-in Haskell monad machinery. Code is in hw5.hs on course web page.

1. [20 points]
   Define type constructor `M`, and function `run` to match the given types.

2. [30 points]
   Define `interp` clauses for `Let` and `LetRec` such that:

   - the declarations are independent
   - there is no variable capture
   - malformed LetRec declarations (i.e. not of form `x :=: v :\ e`) trigger an interpreter error.

3. [50 points]
   Add nondeterminism back into the interpreter:

   - Change `M` and `run` so that
     `test :: Term -> [((Either String Value, String), MyState)]`
     Also change second `instance Show` declararation.
   - Do not change any existing interpreter machinery.
   - Add a `Term` constructor `Amb Term Term` and one `interp` clause for `Amb`, do not use a helper function.