Lecture 2

Jose’s workflow

```
> ghci Basics.hs
ghci> 3 + 5
...
-- to edit a file
-- you need to set the $EDITOR environment variable for this command to work
ghci> :e
```

We are working with Haskell because of its purity - the enforcement of the distinction between pure functions and functions with side effects. This is why this is an interesting language to work with.

Sequencing actions: done with `do` notation. makes it look like youre in an imperative language (do one thing, do the next, etc.), but there are still just expressions, not statements.

```
-- type inference happens in haskell, but for assignments we must
-- declare types.
fun = do
  putStrLn "Hello"
  res <- getContents
  putStrLn res
-- can ask haskell for the type of a function:
ghci> :t fun   -- fun :: IO ()

-- Imagine if we did
fun = do
  putStrLn "Hello"
  res <- getContents
  myStr <- res ++ ", hi"  -- causes an error
  putStrLn res
-- the statement mixes pure values (myStr is a list which is pure) with
-- io actions, which is an error.

-- left arrow means "result of the action" goes into the "variable"
-- but, we said the result of the action `res ++ ", hi"` should go in
-- myStr, but it is not an action!

-- to name pure values, we have a construct to let us name pure values
-- (let expressions)
```

```
fun = do
  putStrLn "Hello"
  res <- getContents
  let myStr = res ++ ", hi"
  putStrLn myStr
```

The haskell language forces the distinction between pure values and impure actions.

Anything that does anything with the computer/requires a system calwould be considered an impure action in haskell.

Using HUnit library:

```
t1 :: Test
t1 = (1 + 2 :: Int) ~?= 3
```

Like an IO action, this does not actually do anything, this just makes a test. We have to run this purposefully.

```
-- we use the action runTestTT :: Test -> IO Counts  to run our test.

numTest :: IO Counts
numTest = runTestTT t1
```

Structured Data

Tuples are the main way to have structured data.

```
-- type is written as (A1, ..., An) where Ai is the type of the value
-- at position i
tup1 :: (Char, Int)
tup1 = ('a', 5)

tup3 :: ((Int, Double), Bool)
tup3 = ((7, 5.2), True)

ghci> :t fst
fst :: (a, b) -> a
ghci> :t snd
snd :: (a, b) -> b

-- to extract values from tuples, you need pattern matching! :D
that :: (Int, Int, Int) -> Int
that (a, b, c) = a * (b + c)
```

```
-- we can put ANYTHING in tuples, even io actions :O
act2 :: (IO (), IO ())
act2 = (putStr "Hello", putStr "World")

-- can do this
let (a, b) = act2

--
-- the left arrow is about running an action, but the tuple act2 is pure,
-- so we use the pure binding let here for the two actions
-- then we can just write x, y to perform the first and second actions.
runAct2 :: IO ()
runAct2 = do
  let (x, y) = act2
  x
  y

runAct2' :: IO ()
runAct2' :: IO()
runAct2' = do
  let (x, y) = act2
  x
  x   -- can do it twice!
```

The analogy Jose gave regarding IO actions is that if you write a recipe, the end result is not the thing you wrote a recipe for, it's just a set of instructions. Similarly, we can make an IO action, but then we must separately run it.

Optional Values

Options are written with `Maybe`

```
-- the Just tag tells compiler that we have a value
m1 :: Maybe Int
m1 = Just 2

-- Nothing tag means we have no value.
m2 :: Maybe Int
m2 = Nothing

ghci> safeDiv x y = if y != 0 then Just (x `div` y :: Int) else Nothing
ghci> :t safeDiv
safeDiv :: Int -> Int -> Maybe Int
```

We can take any function and turn it into an infix operation.

```
ghci> 10 `safeDiv` 5
Just 2
```

we can perform pattern matching when defining a function

```
pat'' :: Maybe Int -> Int
pat'' (Just x) = x
pat'' Nothing = 2

-- patterns can be nested
jn :: Maybe (Maybe a) -> Maybe a
jn (Just (Just x)) = Just x
jn (Just Nothing) = Nothing
jn Nothing = Nothing
```

List

Denoted `[A]`, meaning a list of `A`. Lists must be homogeneous.

```
l1 :: [Double]
l1 = [1.0, 2.0, ...]

-- can contain structured data
l3 :: [(Int, Bool)]
l3 = [(1, True), (2, False)]

-- empty list
l6 :: [a]
l6 = []

-- lists are not arrays !!! we signal end of lists with empty list.
-- "String" is just another name for a list of characters.
l7 :: String
l7 = ['h', 'e', ... '!']

-- cons is the same as in different languages !
cons :: a -> [a] -> [a]
cons = (:) -- can take any operator and partially apply it
-- turns the infix operator (:) into a prefix cons.

c1 :: [Bool]
c1 = True : [False, False]

ghci> True : []
[True]
ghci> (:) True []
```

```
[True]
ghci> (+) 10 14
24
```