Today: a couple of topics

1. monads revisited, and do notation
2. type inference
3. modelling mutation (using types)
(>>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(\>>>=) :: Either String a
   -> (a -> Either String b)
   -> Either String b
class Monad m where
    (>>>=) :: m a -> (a -> m b) -> m b
class Monad m where

  (>>>=) :: m a -> (a -> m b) -> m b

  return :: a -> m a
writing code with chained computations (revisited)
doAll :: Int -> Maybe Int
doAll x =
    case f x of
        Nothing -> Nothing
        Just y ->
            case g y of
                Nothing -> Nothing
                Just z ->
                    case h z of
                        Nothing -> Nothing
                        Just z' -> Just (x + y + length z')
doAll :: Int -> Maybe Int
doAll x =
  case f x of
    Nothing -> Nothing
    Just y ->
      case g y of
        Nothing -> Nothing
        Just z ->
          case h z of
            Nothing -> Nothing
            Just z' -> Just (x + y + length z')
doAll :: Int -> Maybe Int
doAll x =
    f x >>= (\y ->
        g y >>= (\z ->
            h z >>= (\z' ->
                Just (x + y + length z'))))
doAll :: Int -> Maybe Int

doAll x =
  f x >>= \y ->
  g y >>= \z ->
  h z >>= \z' ->
  Just (x + y + length z')
\[
f \::\!\! \text{Int} \rightarrow \text{Maybe\ Int}
g \::\!\! \text{Int} \rightarrow \text{Maybe\ String}
h \::\!\! \text{String} \rightarrow \text{Maybe\ [String]}
\]

doAll :: Int \rightarrow \text{Maybe\ Int}
doAll x = \textbf{do}
y <- f x
z <- g y
z' <- h z
Just (x + y + \text{length\ z'})
type checking: where do types come from?
Main sources for static type information:

1. literal values, builtins ("base cases")
2. human type annotations (e.g., `int x;`)
type inference

the ability of a tool (e.g., compiler) to *statically* determine the types of expressions, *without* human annotations
Main sources for *static* type information:

1. literal values, builtins ("base cases")
2. human type annotations (e.g., `int x;`)
3. how expressions are used in the code
Function calls generate *constraints* on expression types.

“In order for this program to be correct, this must be a ...”
\[ f \ x \ y = (\text{words} \ x) \ !! \ y \]

\[
\text{words} :: \text{String} \rightarrow [\text{String}]
\]

\[
(!!) :: [a] \rightarrow \text{Int} \rightarrow a
\]
\[ f \ x \ y = (\text{words}\ x) \ !\!\! \ y \]

\[
\text{words} :: \text{String} \rightarrow [\text{String}]
\]

\[
(!!) :: [a] \rightarrow \text{Int} \rightarrow a
\]
\[ f \ x \ y = (\text{words } x) !! y \]

```
words :: String -> [String]
(!!) :: [a] -> Int -> a
```

\[ x :: \text{String} \]

\[ \text{String} \]

\[ y :: \text{Int} \]
\[ f \ x \ y = (\text{words } x) \ !! \ y \]

\[
\begin{align*}
\text{words} & : \text{String} \rightarrow [\text{String}] \\
(\text{!!}) & : [a] \rightarrow \text{Int} \rightarrow a
\end{align*}
\]
\[ f \ x \ y = (\text{words} \ x) \ !! \ y \]

\[ \text{words} :: \text{String} \rightarrow [\text{String}] \]

\[ (!!) :: [a] \rightarrow \text{Int} \rightarrow a \]

\[ f :: \text{String} \rightarrow \text{Int} \rightarrow \text{String} \]
Function calls generate *constraints* on expression types.

“In order for this program to be correct, this must be a ...”
Sometimes *no* constraints are generated. This leads to **generic polymorphism**.
\[ f \ x \ y \ z = (\text{words} \ x) !! y \]

\[ f :: \text{String} \to \text{Int} \to \ a \to \text{String} \]
Constraints *between* types
\[ f(x, y, z) = \text{if } x \text{ then } y \text{ else } z \]

branch types must match, but could be any type
\[ f \ x \ y \ z = \text{if} \ x \ \text{then} \ y \ \text{else} \ z \]

branch types must match, but could be any type

\[ f \ :: \ \text{Bool} \rightarrow a \rightarrow a \rightarrow a \]
Sometimes a constraint partially, but not fully, limits the possible type. These are typeclass constraints.
\[ f(x, y, z) = \]
  \[
  \text{if } x == y \\
  \text{then } z \\
  \text{else } z + 1
  \]

Too specific: \( f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)
\[ f \ x \ y \ z = \]
  \begin{align*}
  &\text{if } x == y \\
  &\text{then } z \\
  &\text{else } z + 1
  \end{align*}

Too general: \( f :: a \rightarrow a \rightarrow b \rightarrow b \)
\[ f(x, y, z) = \]
\[
\text{if } x == y \text{ then } z \text{ else } z + 1
\]

Just right: \( f :: (\text{Eq } a, \text{Num } b) \Rightarrow a \rightarrow a \rightarrow b \rightarrow b \)
Extra office hours for A2 (BA4261)

Monday 3:30-6:00
Tuesday 10:00-noon
Mutation!!
data BTree a = Empty
            | Node a Int (BTree a) (BTree a)
Problem: label each node with its position in the tree’s postorder traversal

postOrderLabel :: BTree a -> BTree a
i = 0

def post_order_label(tree):
    if tree.is_empty():
        return
    else:
        post_order_label(tree.left)
        post_order_label(tree.right)
        tree.root.label = i
        i = i + 1
How do we handle the i?
Recall: generic list iteration

\((\text{foldl} \ f \ \text{seed} \ \text{lst})\)

\(\text{acc} = \text{seed}\)

for \(x\) in \(\text{lst}\):
    \(\text{acc} = f(x, \text{acc})\)
Recall: generic list iteration

(foldl \( f \) seed lst)

\[ \text{acc} = \text{seed} \]
\[
\text{for } x \text{ in lst:}
\]
\[
\quad \text{acc} = f(x, \text{acc})
\]
Key idea: model mutating state through function input/output
data State s a = State (s -> (a, s))
get :: State s s
get = State ($ s \rightarrow (s, s))
put :: s -> State s ??
put x = State (\s -> (??, x))
put :: s -> State s ()
put x = State (\s -> (() , x))

() called unit (names both a type and a value).
runState :: State s a -> (s -> (a, s))
runState (State f) = f

-- or...
runState (State f) init = f init
Example!

```c
int x
x = 10
x = x * 2
return x + 1
```
\[
\begin{align*}
(_,\ s1) & = \text{runState} (\text{put } 10) \ s \\
(x,\ s2) & = \text{runState} \ \text{get} \ s1 \\
(_,\ s3) & = \text{runState} (\text{put } (x \times 2)) \ s2 \\
(x',\ s4) & = \text{runState} \ \text{get} \ s3 \\
(x' + 1,\ s4) &
\end{align*}
\]
\[(_, \; s1) \; = \; \text{runState} \; \text{(put} \; 10) \; s\]
\[(x, \; s2) \; = \; \text{runState} \; \text{get} \; s1\]
\[(_, \; s3) \; = \; \text{runState} \; \text{(put} \; (x \; \times \; 2)) \; s2\]
\[(x', \; s4) \; = \; \text{runState} \; \text{get} \; s3\]

\[(x' \; + \; 1, \; s4)\]
Need to sequence stateful operations, using values from previous operations in future ones.

“lookup x, then use it to compute a new value and store it”
Of course it’s a monad.

(>>=) :: State s a -> (a -> State s b) -> State s b
Back to trees!
i = 0

def post_order_label(tree):
    if tree.is_empty():
        return
    else:
        post_order_label(tree.left)
        post_order_label(tree.right)
        tree.root.label = i
    i = i + 1