Wrapping up State: post-order labelling
From a mutable int to a mutable environment

In the first part of the course, we saw an interpreter that used an environment as an explicit parameter:

(interpret expr env)

We can model the same kind of behaviour using State:

interpret :: Expr -> State (Map String a) a
Talking to the outside world: **IO**

“If a program runs without any side effects, does it make a sound?”
Key idea: treat the outside world as some big mutable State
Standard input, standard output

getLine :: World -> (String, World)
print :: String -> World -> ((), World)
File I/O

open :: Path -> World -> (FILE *, World)
read :: FILE * -> World -> (String, World)
Subprocesses

run :: [String] -> World -> (PID, World)
kill :: PID -> World -> (() , World)
Network requests

checkFacebook :: World -> (😊, World)
submitWork :: Assignment -> World -> (⭐, World)
State World a = State (World -> (a, World))
State World a = State (World → (a, World))

IO a
A value of type `IO a` represents an I/O action that produces a value of type `a`.

An I/O action is **only** performed when:
1. We evaluate it in the interpreter.
2. We execute it inside `main (main :: IO ())`. 
Standard input, standard output (for real)

getLine :: IO String
putStrLn :: String -> IO ()
File I/O (for real)

openFile :: FilePath -> IOMode -> IO Handle
hGetContents :: Handle -> IO String
hClose :: Handle -> IO ()
Reminder: next week, we have lectures on Monday and Wednesday (both in the regular lecture room).
IO is the ultimate *lack of constraints*

main :: IO () --- main can do anything!
IO is the ultimate *lack of constraints*

main :: IO () --- main can do anything!
Recall this “identity-ish” function...

```java
<T> T f(T x) {
    sendDavidSpam();
    return x;
}
```
In Haskell, sending David spam is an IO action...

```haskell
f :: a -> IO a
f x = do
  _ <- sendDavidSpam
  return x
```
A fundamental design principle of a Haskell program is to have an impure shell surrounding an pure core.

```
analyzeDataFile :: FilePath -> IO ___
analyzeDataFile path = do
  handle <- openFile path ReadMode
  text <- hGetContents handle -- text is a String
  let result = processData text -- result is a ___
  _ <- hClose handle
  return result

processData :: String -> ___
```
Type checking: where do types come from?
Main sources for **static** type information:

1. literal values, built-ins
2. type annotations (e.g., `int x;`)
type inference

the act of determining the type of expressions in a program, statically and without annotations
Main sources for **static** type information:

1. literal values, built-ins
2. type annotations (e.g., `int x;`)
3. how expressions are used
We’ve already seen how expression types generate constraints on how expressions can be used.

“If x is a Bool, then (x && True) is valid.”

“If x is not a Bool, then (x && True) is not valid.”
But how expressions are used also generate constraints on expression types!

“If \((x \&\& \text{True})\) is valid, then \(x\) is a \(\text{Bool}\).”
\[
f(x, y) = (\text{words } x) !! y
\]
f x y = (\textbf{words} x) !! y

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

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
\text{words} & : \text{String} \to [\text{String}] \\
(\text{!!}) & : [\text{a}] \to \text{Int} \to \text{a}
\end{align*}
\]

\[
\begin{align*}
f & : \text{String} \to \text{String} \\
x & : \text{String} \\
y & : \text{Int}
\end{align*}
\]
\[
\text{words} :: \text{String} \to [	ext{String}]

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

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

\[
f :: \text{String} \to \text{Int} \to \text{String}
\]
Sometimes *no* constraints are generated for an expression. This leads to **generic polymorphism**.
\[
f \ x \ y \ z = (\text{words } x) \ !! \ y
\]

\[
f :: \text{String} \rightarrow \text{Int} \rightarrow \text{a} \rightarrow \text{String}
\]
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
Typeclass constraints are generated by using their member functions.
\[ f \ x \ y \ z = \]
\[
  \text{if } x == y \text{ then } z \]
\[
  \text{else } z + 1
\]

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

Just right: \( f :: (\text{Eq } a, \text{Num } b) \Rightarrow a \rightarrow a \rightarrow b \rightarrow b \)