CS 344 - Higher Order Functions

- HW due Monday in class
- Next HW still due next Friday
Where Bindings

We can set local names for expressions.

Ex:

\[
\text{initials} :: \text{String} \to \text{String} \to \text{String} \\
\text{initials} \ f \ \text{firstname} \ \text{lastname} = \\
[f] \ + \ . \ + \ [l] \ + \ .
\]

where \( f \) = \text{firstname}
\( l \) = \text{lastname}
Let bindings

let <bindings> in <expression>

> let square x = x * x in (square 5, square 3, square 2)

\[ \rightarrow (25, 9, 4) \]

Difference: lets are expressions themselves (with just gives a local nickname)

Ex: \[ \rightarrow 4 \ast (\text{let } a = 9 \text{ in } a + 1) + 2 \]

\[ \rightarrow 42 \]
Ex: function to compute area of a cylinder
Recursion in Haskell

How do we find the maximum element in a list?

Sequential (or linear) search

\[
\begin{align*}
\text{maximum'} &: \quad \text{[ ] } = \text{error} \\
\text{maximum'} &: \quad [x] = x \\
\text{maximum'} &: \quad (x : xs) = \begin{cases} 
  x & \text{if } x \gt \text{max}\text{tail} \\
  \text{max}\text{tail} & \text{otherwise}
\end{cases}
\end{align*}
\]

How do we do it recursively?

\[
\text{max} (\text{list}) \rightarrow \text{returns max element}
\]
\text{maximum'} (x:xs) = \text{max} x (\text{maximum'} xs)
How about reversing a list?

Code this one yourself, a
think recursively.

\[
\text{Start:} \\
\text{rev:} : \ [a] \rightarrow \ [a]
\]
Higher Order Functions

Haskell functions can take other functions as input parameters.

In fact, we've been doing this already.

Why? Haskell functions can only have one input parameter!

\[
\text{max}(4, 5) \uparrow
\]

Space is a function application (it has highest precedence)
So: max 4 returns a function which returns either 4 or the input parameter.

Really, have (max 4) 5 -

Check: \[ \text{let max} \]
\[ \langle \text{ord a} \rangle \Rightarrow a \rightarrow (a \rightarrow a) \]

\[ \text{return type} \]
Some simple applications

Look at `multThree.hs`

A simple function to multiply 3 numbers together.

```haskell
> let multTwoWith6 = multThree 6
> let multWith12 = multTwoWith6 2
```
Higher Order functions (cont)

So by calling a function w/ not enough parameters, we're creating functions as we go.

Note: try without a let

```haskell
data Mult Three 3 4
```

Produces a function, which isn't part of the Show type class, so Haskell can't print it.
Infix functions: use ( )

Ex: divideByTen :: (Floating a) => a -> a
    divideByTen = (/10)

divideByTen 200
    → 20

Ex: isUpper :: Char -> Bool
    isUpper = (elem) [A..'Z']
Now: functions as parameters

applyTwice example:

Note: - parenthesis in type are now mandatory! Why?

Ex: > applyTwice (+3) 10
    > applyTwice ("HAHA " ++) "HEY"
    > applyTwice (multThree 2 2) 10
    > applyTwice (3:) [1]