Consolidation & Homeworaks

Yan Huang
Goal for Today

• Consolidate your learning from the past few lectures
• HW2
List Comprehension and **zip**
Using “zip” we can define a function that returns the list of all positions of a value in a list:

\[
\text{positions :: Eq a \Rightarrow a \rightarrow [a] \rightarrow [Int]}
\]

For example:

\[
> \text{positions 0 [1,0,0,1,0,1,1,0]} \rightarrow [1,2,4,7]
\]
List Comprehension and \texttt{zip}

\begin{verbatim}
positions :: Eq a \Rightarrow a \rightarrow [a] \rightarrow [Int]
\end{verbatim}
Using zip we can define a function that returns the list of all positions of a value in a list:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
    [i | (x’,i) ← zip xs [0..], x == x’]
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```
String Comprehensions

A string is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

"abc" :: String

Means ['a', 'b', 'c'] :: [Char].
Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```excel
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```
Similarly, list comprehensions can also be used to define functions on strings, such as counting how many times a character occurs in a string:

```haskell
count :: Char -> String -> Int

count c s = length [i | i <- s, i == c]
```

For example:

```haskell
> count 's' "Mississippi"
4
```
Similarly, list comprehensions can also be used to define functions on strings, such as counting how many times a character occurs in a string:

\[
\text{count} :: \text{Char} \rightarrow \text{String} \rightarrow \text{Int} \\
\text{count} \ x \ xs = \\
\quad \text{length} \ [x' \mid x' \leftarrow xs, x == x']
\]

For example:

\[\\> \text{count} \ 's' \ "Mississippi"\\= 4\\]
Recursive Functions and Quick Sort
Quick Sort

\[
\begin{align*}
5 & \quad 7 \quad 8 \quad 1 \quad 2 \quad 6 \quad 9 \\
\{1, 2\} & \quad \{7, 8, 6, 9\} \\
1 & \quad 2 \quad 5 \quad 6 \quad 7 \quad \{8, 9\} \\
1 & \quad 2 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9
\end{align*}
\]
Quick Sort

The quicksort algorithm for sorting a list of values can be specified by the following two rules:

1. The empty list is already sorted;
2. Non-empty lists can be sorted by sorting the tail values \( \leq \) the head, sorting the tail values \( > \) the head, and then appending the resulting lists on either side of the head value.
Using recursion, this specification can be translated directly into an implementation:
Using recursion, this specification can be translated directly into an implementation:

```haskell
qsort :: Ord a ⇒ [a] → [a]
qsort [] = []
qsort (x:xs) =
  qsort smaller ++ [x] ++ qsort larger
where
  smaller = [a | a ← xs, a ≤ x]
  larger  = [b | b ← xs, b > x]
```

This is probably the simplest implementation of quicksort in any programming language!
For example (abbreviating qsort as q):

\[ qsort \{3,2,4,1,5\} \]

\[ qsort \{2,1\} ++ [3] ++ qsort \{4,5\} \]

\[ qsort \{1\} ++ [2] ++ qsort [] \]


\[ [1] \]

\[ [] \]

\[ [] \]

\[ [5] \]
Homework 2
(\text{\texttt{a}}, \text{\texttt{b}}, \text{\texttt{c}})  
\text{\texttt{char}, char, char}

\text{\texttt{type}} (\text{\texttt{a}}, \text{\texttt{b}}, \text{\texttt{c}})

\text{\texttt{swap} :: (a, b) \rightarrow (b, a).}
\text{\texttt{swap} (x, y) = (y, x)}
\( f : (\mathbb{R}^2, +, \cdot) \rightarrow \text{Rational} \)

\( f \ x = \text{forRational } \& \text{ Rem } x \ 5 \)

\[
\text{dotprod} \ [1, 2, 3] \ [5, 6, 9] = 1 \times 5 + 2 \times 6 + 3 \times 9
\]

\[
\sum_{(x,y) \in \text{zip } x \ 5 \ y} \text{dotprod } x 5 \ y 5
\]