Today

- Quiz 6
- More on partial functions
- More list functions

Assignments

- R-7, HW-7 due
- R-8, HW-8 out (due Tues)
Partial function application

Allows us to define **partial applications** of the function

```
Prelude> let take4 = take 4

Prelude> :t take
take :: Int -> [a] -> [a]

Prelude> :t take4
take4 :: [a] -> [a]

Prelude> take4 [1, 3 .. 21]
[1, 3, 5, 7]
```

- Where `take` and `take4` have the types

```
take :: Int -> [a] -> [a]
take4 :: [a] -> [a]
```

Exercise 2
Lambda (i.e., unnamed, anonymous) Functions

Frequently used to define functions “on the fly”

Prelude> (\x -> x * 2) 4
 8

Prelude> (\x y -> x + y) 3 4
 7

Prelude> filter (\x -> x < 5) [1..10]
[1,2,3,4]

You can think of partial functions as creating/returning lambda functions

Prelude> let add x y = x + y

Prelude> :t add
add :: (Num a) => a -> a -> a

Prelude> :t add 2 -- a partial function
add 2 :: (Num t) => t -> t

Prelude> :t (\y -> 2 + y) -- the lambda equivalent
(\y -> 2 + y) :: (Num t) => t -> t

Here (add 2) returns the lambda function (\y -> 2 + y)
Partial application is supported by “currying”

- all functions take one argument
- functions can return values or functions
- so really:
  \[(\lambda x\ y \rightarrow x + y)\]  -- similar to: \[\text{add } x\ y = x + y\]

- is this:
  \[(\lambda x \rightarrow (\lambda y \rightarrow x + y))\]  -- similar to: \[\text{add } x = (\lambda y \rightarrow x + y)\]

- currying happens for all multi-argument functions in Haskell
More Haskell List Functions

List \textbf{length} gives number of elements in a list

\begin{verbatim}
Prelude> length [1..5]
5

Prelude> length []
0
\end{verbatim}

List \textbf{init} gives list minus last value

\begin{verbatim}
Prelude> init [4, 1, 5, 3]
[4, 1, 5]

Prelude> init [1]
[]

Prelude> init []
*** Exception: Prelude.init: empty list
\end{verbatim}

List \textbf{last} gives last element in list

\begin{verbatim}
Prelude> last [4, 1, 5, 3]
3

Prelude> last []
*** Exception: Prelude.last: empty list
\end{verbatim}
List reverse gives list reversed

Prelude> reverse [4, 1, 5, 3]
[3, 5, 1, 4]

Prelude> reverse [1]
[1]

Prelude> reverse []
[]

List take n gives first n elements as sublist

Prelude> take 2 [4, 1, 5, 3]
[4, 1]

Prelude> take 1 [4, 1, 5, 3]
[4]

Prelude> take 0 [4, 1, 5, 3]
[]

Prelude> take 5 [4, 1, 5, 3]
[4, 1, 5, 3]

Prelude> take (-1) [4, 1, 5, 3]
[]

Exercise 1
List drop \( n \) gives list minus first \( n \) elements

Prelude> drop 2 [4, 1, 5, 3]
[5, 3]

Prelude> drop 1 [4, 1, 5, 3]
[1, 5, 3]

Prelude> drop 5 [4, 1, 5, 3]
[]

Exercise 2 and 3

List replicate \( n \) \( v \) returns a list of \( n \) values \( v \)

Prelude> replicate 5 1
[1,1,1,1,1]

List index \((!!)\) returns the value at the given index

Prelude> [1..5] !! 0
1

Prelude> [1..5] !! 2
3

List zip takes two lists and pairs their elements

Prelude> zip [1,2] [3,4]
[(1,3),(2,4)]

Prelude> zip [1,2,3] [4,5]
[(1,4),(2,5)]
List `elem x` is true if `x` in list

```
Prelude> elem 1 [4, 1, 5, 3]
True

Prelude> elem 0 [4, 1, 5, 3]
False

Prelude> elem 'a' [4, 1, 5, 3]
<interactive>:15:11:
  No instance for (Num Char)
  arising from the literal ‘4’
  ...
```

List `notElem x` is false if `x` in list

Higher-order function `filter` removes elements of a list

```
Prelude> filter even [1..10]
[2,4,6,8,10]
```

- We’ll talk about more functions on lists later ...
The error function

- Useful for “error” cases
- Aborts execution (exception) without returning a value

Example:

```haskell
secondElem xs = if length xs >= 2
              then head (tail xs)
              else error "list too short"
```

> secondElem [1,2]
2

> secondElem [1]
*** Exception: list too short

Q: What is the type of the error function?

> :t error
error :: [Char] -> a

- Given a string returns a value of any type a

Q: Why does error return any type?

- Always returns a value of the “correct” type
- Thus, can be called from anywhere, without causing a type error
- (Again, never returns though ... throws an exception)