Today

- Recursion
- Pattern Matching

Assignments

- HW8 due Thurs
- HW9 out on Thurs
**Warm Up: More Functions and Types**

`replicate n v` returns a list of `n` values `v`

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

Q: What is the type of `replicate`?

```haskell
(!!) returns the value at the given index
```

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

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

Q: What is the type of `(!!)`?

`zip` takes two lists and pairs their elements

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

Prelude> zip [1,2,3] [4,5]
[(1,4),(2,5)]
```

Q: What is the type of `zip`?
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'
  ...

Q: What is the type of elem?

Higher-order function filter removes elements of a list

Prelude> :t even
even :: Integral a => a -> Bool

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

• we’ll talk more about HOF types soon
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"
```

```haskell
> secondElem [1,2]
2

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

Q: What is the type of the error function?

```haskell
> :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)
More Function Types: Higher order functions

Parentheses are important in function types

```haskell
Prelude> any even [1, 3 .. 11]
False
```

```haskell
Prelude> :type any
any :: (a -> Bool) -> [a] -> Bool
```

• The first argument is a function from \( a \) to \( \text{Bool} \)^1

For example

```haskell
Prelude> let any1 = any 1
... No instance for (Num (a -> Bool)) arising from the literal '1'
```

• Why doesn't \( \text{any1} \) work? ... 1 is not a function from \( a \) to \( \text{Bool} \)

Another example

```haskell
Prelude> :type even
even :: (Integral a) => a -> Bool
```

```haskell
Prelude> let anyEven = any even
```

• This works! ... why? (Hint: what is the type of \( \text{even} \)?)

---

^1 Note that the type is different in newer versions of Haskell ... more later
More examples of higher order functions

The map function applies a function to each element of a list

Prelude> map even [1,2,3,4]
[False,True,False,True]

Prelude> map (+1) [1,2,3,4]
[2,3,4,5]

Q: What is the type of map?

The zipWith function applies a function to each zip pair

Prelude> zipWith (+) [1,2,3] [10,20,30]
[11,22,33]

Prelude> zipWith (<) [1,2,3] [10,20,30]
[True,True,True]

Q: What is the type of zipWith?

We'll talk about how these work, and more hof's later
Recursion

Sum the elements of a list

1. Sum of the empty list is 0 ... “Base Case”

2. Sum of non-empty list is first elem plus sum of rest ... “Recursive Step”

One way to write this in Haskell

\[
\text{mySum } xs = \begin{cases} 
0 & \text{if } \text{null } xs \\
\text{head } xs + \text{mySum } (\text{tail } xs) & \text{otherwise}
\end{cases}
\]

How it works

\[
\text{mySum } [2, 4, 6] \implies 2 + \text{mySum } [4, 6] \implies 2 + 4 + \text{mySum } [6] \\
\implies 2 + 4 + 6 + \text{mySum } [] \implies 2 + 4 + 6 + 0
\]

Q: What is the type of mySum?

\[
\text{Prelude}\gg \text{:type mySum} \\
\text{mySum} :: (\text{Num } a) \Rightarrow [a] \rightarrow a
\]

Q: How does Haskell know this?

- by looking at the function definition!
- for example:

\[
- \text{null} :: [a] \rightarrow \text{Bool} \quad \text{... xs must be a list}
- 0 :: \text{Num } a \Rightarrow a \quad \text{... return type is a Num instance}
- \text{head} :: [a] \rightarrow a \quad \text{... xs element}
- (+) :: (\text{Num } a) \Rightarrow a \rightarrow a \quad \text{... list elem types a Num instance}
\]
More Examples

Drop

Recall that \texttt{drop n xs} returns a list with first \texttt{n} elems of \texttt{xs} removed.

\begin{verbatim}
Prelude> drop 2 "foobar"
"obar"

Prelude> drop 4 "foobar"
"ar"

Prelude> drop 4 [1,2]
[]

Prelude> drop 0 [1, 2]
[1, 2]

Prelude> drop 7 []
[]

Prelude> drop (-2) "foo"
"foo"
\end{verbatim}

Q: What are the base cases?

- Empty list (nothing to drop)
- \texttt{n} \leq 0 (nothing to drop)
Q: How can we define our own version of \texttt{drop}?

$$\text{myDrop } n \;xs = \begin{cases} xs & \text{if } n \leq 0 \text{ || } \text{null } xs \\ \text{myDrop } (n-1) \; (\text{tail } xs) & \text{else} \end{cases}$$

- if either base case, return list
- otherwise, call \texttt{myDrop} on smaller \texttt{n} and smaller \texttt{xs}

Q: What is the type of \texttt{myDrop}?

- Hint 1: \texttt{(<=) :: (Ord a) => a -> Bool}
- Hint 2: \texttt{(-) :: (Num a) => a -> a -> a}

$$\text{myDrop} :: (\text{Num a, Ord a}) => a -> [b] -> [b]$$