Today

- Quiz 8
- Function types (cont)
- Recursion

Assignments

- HW 8 due
- HW 9 out
- Proj Status Update due Tues
- Exam 2 next Thurs
More Function Types: Higher order functions

Parentheses are important in function types

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

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

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

For example

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

• Why doesn't any1 work? ... 1 is not a function from a to Bool

Another example

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

Prelude> let anyEven = any even

• This works! ... why? (Hint: what is the type of 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

```haskell
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

```haskell
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

Basic idea (and similar to inductive proofs):

- Solve a problem by breaking the problem into smaller (identical) subproblems
- Each subproblem makes a little bit of progress
- But taken together, end up solving the big problem

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

```haskell
mySum xs = if null xs
    then 0
    else head xs + mySum (tail xs)
```

How it works

```
mySum [2,4,6]
  ==> 2 + mySum [4,6]
  ==> 2 + 4 + mySum [6]
  ==> 2 + 4 + 6 + mySum []
  ==> 2 + 4 + 6 + 0
```

Q: What is the type of mySum?

```
Prelude> :type mySum
mySum :: (Num a) => [a] -> a
```
Q: How does Haskell know this?

• by looking at the function definition!

• for example:

  - null :: [a] -> Bool  ... xs must be a list
  - 0 :: Num a => a     ... return type is a Num instance
  - head :: [a] -> a    ... xs element
  - (+) :: (Num a) => a -> a  ... 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?

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

\begin{verbatim}
myDrop n xs = if n &le; 0 || null xs then xs
        else myDrop (n-1) (tail xs)
\end{verbatim}

- 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: \((\leq) :: (\text{Ord } a) \Rightarrow a \Rightarrow \text{Bool}\)
- Hint 2: \((-) :: (\text{Num } a) \Rightarrow a \Rightarrow a \Rightarrow a\)

\begin{verbatim}
myDrop :: (\text{Num } a, \text{Ord } a) \Rightarrow a \Rightarrow [b] \Rightarrow [b]
\end{verbatim}
Exponentiation (Power)

Q: Using recursion, define a function \texttt{pow x y} that computes \(x^y\) for \(y \geq 0\)

\[
\text{pow x y} = \begin{cases} 
1 & \text{if } y = 0 \\
\text{else } x \times \text{pow } x \ (y-1) & \text{otherwise}
\end{cases}
\]

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

- \texttt{Hint 1: (*) :: (Num } a \texttt{) => a -> a -> a}

\[
\text{pow :: (Num } a \texttt{, Num } b \texttt{, Eq } b \texttt{) => a -> b -> a}
\]

Append

Q: Using recursion, define a function \texttt{concat xs ys} that computes \(xs ++ ys\)

\[
\text{concat xs ys} = \begin{cases} 
ys & \text{if } \text{null } xs \\
\text{else } (\text{head } xs) : \text{concat } (\text{tail } xs) \ ys & \text{otherwise}
\end{cases}
\]

The basic idea:

- appending an empty list to \(ys\) is just \(ys\) (base case)
- otherwise, create the new list by:
  - adding head of \(xs\) to the result of a smaller append
  - which simply appends \(ys\) to the tail of \(xs\)
How it works: w/out laziness

```
append [1,2] [3,4]
  ==> 1 : append [2] [3,4]
  ==> 1 : 2 : append [] [3,4]
  ==> 1 : 2 : [3,4]
  ==> [1,2,3,4]
```

Another example: w/ laziness

```
append [1,2] [3..]
  ==> (head [1,2]) : append (tail [1,2]) [3..])
  ==> (head [1,2]) : (head [2]) : append (tail [2]) [3..]
  ==> (head [1,2]) : (head [2]) : [3..]
```

- second list never has to (is never asked to) be evaluated
- result is called a “thunk” (“suspension”, “delayed computation”, “future”)

Why does `append` have the type `[a] -> [a] -> [a]`?

- `null :: [a] -> Bool  ... xs :: [a]`
- `(:) :: b -> [b] -> [b]  ... result must be of type [b]`
- `then ys  ... ys must of type [b]`
- `head xs :: a  ... so a = b`