Lecture 37:

- More on function types

Announcements:

- HW-7 due
- HW-8 out (soon)
Function types

Functions have types (either given or inferred)

Prelude> not True
False

Prelude> :type not
not :: Bool -> Bool

• The -> is read as “to” or “returns”

“not has the type Bool to Bool”

“not takes a Bool and returns a Bool”

Another example

Prelude> succ 6
7

Prelude> :type succ
succ :: Enum a => a -> a

• Here we have a class constraint

“for all Enum types a, succ has the type a to a”
And another example

Prelude> head [1..4]
1

Prelude> :type head
head :: [a] -> a

• Note no class constraint on type a

• This means a is a simple type variable

  “head has the type list of any type a to a”

• Type variables must begin with a lowercase letter

• Whereas types (and type classes) must always be capitalized
Functions with multiple arguments

Example

Prelude> take 4 [1, 3 .. 21]
[1, 3, 5, 7]

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

For now you can view the function as ...

• Having an argument of each type preceding the last ->
• Having the return type following the last ->
• Here: take receives an Int and [a] and returns an [a]

But why two -> (to) symbols?

• -> always denotes a function that ...
  1. takes an argument of the type on the left and
  2. returns the type on the right
• So here, the type on the right ([a] -> [a]) is a function!
  – That is, take 4 returns a function from a list of a to a list of a
• -> is right-associative: a -> a -> a == a -> (a -> a)
**Partial function application**

Allows us to define partial applications of the function

```haskell
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

  ```haskell
  take :: Int -> [a] -> [a]
take4 :: [a] -> [a]
  ```
More Haskell List Functions

length gives number of elements in a list

Prelude> length [1..5]
5

Prelude> length []
0

Q: What is the type of length?

init gives list minus last value

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

Prelude> init [1]
[]

Prelude> init []
*** Exception: Prelude.init: empty list

Q: What is the type of init?
last gives last element in list

```
Prelude> last [4, 1, 5, 3]
3

Prelude> last []
*** Exception: Prelude.last: empty list
```

Q: What is the type of `last`?

reverse gives list reversed

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

Prelude> reverse [1]
[1]

Prelude> reverse []
[]
```

Q: What is the type of `reverse`?
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]
[]

Q: What is the type of take?

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]
[]

Q: What is the type of drop?
(!!) returns the value at the given index

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

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

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

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)]
```

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 \texttt{elem}?

Higher-order function \texttt{filter} removes elements of a list

Prelude> :t even
even :: Integral a => a \rightarrow Bool

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

$\bullet$ we'll talk more about HOF types soon
The error function

- Aborts execution (exception) without returning a value

Example:

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)
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`?