Today

- Function Types

Assignments

- HW-3, R-3 due
- HW-4, R-4 out

Announcements

- Quiz 3 on Thurs
- Project status report next Thurs (11th)
Function types

Functions have types (either given or inferred)

Prelude> not True
False

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

• The \(-\rightarrow\) 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

\[
\text{Prelude} \text{> take 4 [1, 3 .. 21]}
\]  
[1, 3, 5, 7]

\[
\text{Prelude} \text{> :type take}
\]
\[
\text{take} :: \text{Int} \to \text{[\text{a}]} \to \text{[\text{a}]}
\]

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: \text{take} receives an \text{Int} and \text{[\text{a}]} and returns an \text{[\text{a}]}

But why two -> (to) symbols?

- \text{->} 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 (\text{[\text{a}]} \to \text{[\text{a}]}) is a function!
  - That is, \text{take 4} returns a function from a list of \text{a} to a list of \text{a}
- \text{->} is right-associative: \text{a \to a \to a} == \text{a \to (a \to a)}

Exercise 1
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) Functions

Frequently used to define functions “on the fly”

\[ \text{Prelude}\> (\lambda x \rightarrow x \times 2)\> 4 \]
\[ 8 \]

\[ \text{Prelude}\> (\lambda x \ y \rightarrow x \ + \ y)\> 3 \ 4 \]
\[ 7 \]

\[ \text{Prelude}\> \text{filter} \> (\lambda x \rightarrow x < 5) \> [1..10] \]
\[ [1,2,3,4] \]

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

\[ \text{Prelude}\> \text{let} \> \text{add} \> x \ y = x \ + \ y \]

\[ \text{Prelude}\> \text{:t} \> \text{add} \]
\[ \text{add} :: (\text{Num} \ a) => a \rightarrow a \rightarrow a \]

\[ \text{Prelude}\> \text{:t} \> \text{add} \> 2 \> \text{-- a partial function} \]
\[ \text{add} \> 2 :: (\text{Num} \ t) => t \rightarrow t \rightarrow t \]

\[ \text{Prelude}\> \text{:t} \> (\lambda y \rightarrow 2 + y) \> \text{-- the lambda equivalent} \]
\[ (\lambda y \rightarrow 2 + y) :: (\text{Num} \ t) => t \rightarrow t \]

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

- all functions take one argument
- functions can return values or functions
- so really:

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

- is this:

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

- currying happens for all multi-argument functions in Haskell