Today

- Function Types (cont)
- More on Functions

Assignments

- HW 8 out (due Tues)
- Proj Status Update due Apr 9
- Exam 2 Apr 11
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)
Exercise

1. What are the types of the following functions?
   a. `tail ::`
   b. `null ::`
   c. `(++) ::` ... list concatenation
   d. `(:) ::` ... list construction
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

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

2. What are the types of the following partial functions?
   
a. (++) ['a','b'] ::

b. (:) 1 ::

The `replicate` function takes a number \( n \) (Int) and a value \( v \), and returns a list with \( n \) copies of \( v \).

```
Prelude> replicate 4 'a'
['a','a','a','a']

Prelude> replicate 3 4
[4,4,4]
```

What are the types of the following?

c. `replicate` ::

d. `replicate 5` ::
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:

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

- is this:

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

- currying happens for all multi-argument functions in Haskell
- recall \( \lambda \)-calculus

Note: you can do “pattern matching” with lambda functions:

... we’ve only seen this for accessing elements in tuples

-- no pattern matching
pairPred p = (pred (fst p), pred (snd p))

-- w/ pattern matching
pairPred (x, y) = (pred x, pred y)

-- w/ lambda function
pairPred = (\((x,y) \rightarrow (\text{pred } x, \text{pred } y))

-- another example (more on filter later ...)
filter (\((x,y) \rightarrow x > 0 \ \&\& \ y > 0) [(0,1), (1,0), (2,1)]
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?
Exercise

1. Define \texttt{tail} using \texttt{drop}.

\begin{verbatim}
  tail :: [a] -> [a]
  tail xs = drop 1 xs
\end{verbatim}

2. Write a function \texttt{shave} that uses \texttt{take} and \texttt{drop} to “shave off” \( n \) elements from the front and back of a list. For example, \texttt{shave 2 [1..10]} should return the list \([3, 4, 5, 6, 7, 8]\).

\begin{verbatim}
  shave :: Int -> [a] -> [a]
  shave n xs = drop 2 (take ((length xs) - n) xs)

  -- or --

  shave n xs = take ((length xs) - 2*n) (drop n xs)
\end{verbatim}
replicate \( n \) \( v \) returns a list of \( n \) values \( v \)

\[
Prelude> \text{replicate 5 1} \\
[1,1,1,1,1]
\]

Q: What is the type of replicate?

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

\[
Prelude> \text{[1..5] !! 0} \\
1
\]

\[
Prelude> \text{[1..5] !! 2} \\
3
\]

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

zip takes two lists and pairs their elements

\[
Prelude> \text{zip [1,2] [3,4]} \\
[(1,3),(2,4)]
\]

\[
Prelude> \text{zip [1,2,3] [4,5]} \\
[(1,4),(2,5)]
\]

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 about higher-order function types 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"
```

```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)