Today
  - More on Defining Haskell Functions

Assignments
  - HW8 out (due Thur)
Basic Haskell Types (Revisited)

Char
  - Represents (Unicode) characters (e.g., `a')

Bool
  - Represents Boolean values: True or False

Int
  - Signed, fixed-width integer values
  - Size depends on system (today 32 or 64 bits wide)
  - Other smaller numeric types available as well

Integer
  - A signed integer of unbounded size

Double
  - 64 bit floating point numbers (native system representation)
  - Also a Float type, but not used often (smaller, but slower)

Haskell Type Classes

As we've seen, types are more complicated for numbers ...

Prelude> :type 49
49 :: (Num a) => a

- 49 has type a such that a is a member of typeclass Num
- In other words, 49 can be any type that is a member of the Num typeclass
A **typeclass**
- Defines a set of functions (like interfaces in Java or abstract classes in C++)
- Members (types) of the typeclass implement each function

A **typeclass is not the same as a class in C++ or Java**
- In C++/Java class instances are **objects**
- Typeclass instances are **types**

**Some Example Typeclasses**

**Eq** ... types that support equality testing
- All standard Haskell types are members of Eq (except functions and IO)

**Ord** ... types with ordering (e.g., <, >, min, max)
- To be in Ord must be in Eq

**Show** ... types that can be displayed as strings
- Supports the show function (e.g., show 1 returns “1”)

**Read** ... opposite of Show
- Supports the read function
- E.g.: (read "1" :: Int) + 5 returns 6

**Enum** ... types whose values are sequentially ordered
- Functions succ, pred, etc.
- Values used in list enumerations (such as ['a' .. 'z']

S. Bowers 3 of 16
Num

- Functions: +, *, -, negate, etc.
- Integer, Int, Float, Double are instances

Integral ...whole number types (Int and Integer)

- Functions: mod, quot (integer division), ...
- Integer and Int are instances
- Must be of type Real and Enum

Bounded, Floating, Fractional, Real, RealFrac

- To find out about these type :info Fractional, etc., in ghci
- See also the Prelude doc

Called “Class constraints”

For this type ...

\[
\text{Prelude}\geq :\text{type} 49 \\
49 :: (\text{Num}\ a) \Rightarrow a
\]

- Everything before the => is called a **class constraint**
- Only constrains type a to be a member of the Num typeclass
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

  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 \( n \) (\textbf{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:
  \[(\x \ y \to \ x + y)\] -- similar to: \[\text{add } x \ y = x + y\]

- is this:
  \[(\x \to (\y \to x + y))\] -- similar to: \[\text{add } x = (\y \to x + y)\]

- 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

\[
\begin{align*}
\text{-- no pattern matching} \\
\text{pairPred } p &= (\text{pred } \text{fst } p, \text{pred } \text{snd } p)
\end{align*}
\]

\[
\begin{align*}
\text{-- w/ pattern matching} \\
\text{pairPred } (x, y) &= (\text{pred } x, \text{pred } y)
\end{align*}
\]

\[
\begin{align*}
\text{-- w/ lambda function} \\
\text{pairPred} &= (\lambda (x,y) \to (\text{pred } x, \text{pred } y))
\end{align*}
\]

\[
\begin{align*}
\text{-- another example (more on filter later ...)} \\
\text{filter } (\lambda (x,y) \to x > 0 \&\& y > 0) &= [(0,1), (1,0), (2,1)]
\end{align*}
\]
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

\[ \text{Prelude}\> \text{take} \ 2 \ [4, \ 1, \ 5, \ 3] \]
\[ [4, \ 1] \]

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

\[ \text{Prelude}\> \text{take} \ 0 \ [4, \ 1, \ 5, \ 3] \]
\[ [] \]

\[ \text{Prelude}\> \text{take} \ (-1) \ [4, \ 1, \ 5, \ 3] \]
\[ [] \]

Q: What is the type of \text{take}?

drop \( n \) gives list minus first \( n \) elements

\[ \text{Prelude}\> \text{drop} \ 2 \ [4, \ 1, \ 5, \ 3] \]
\[ [5, \ 3] \]

\[ \text{Prelude}\> \text{drop} \ 1 \ [4, \ 1, \ 5, \ 3] \]
\[ [1, \ 5, \ 3] \]

\[ \text{Prelude}\> \text{drop} \ 5 \ [4, \ 1, \ 5, \ 3] \]
\[ [] \]

Q: What is the type of \text{drop}?
Exercise

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

\[
\textbf{tail} :: \texttt{[a]} \rightarrow \texttt{[a]} \\
\text{tail } \texttt{xs} = \texttt{drop 1 } \texttt{xs}
\]

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 \([3,4,5,6,7,8]\).

\[
\textbf{shave} :: \texttt{Int} \rightarrow \texttt{[a]} \rightarrow \texttt{[a]} \\
\text{shave } \texttt{n } \texttt{xs} = \texttt{drop 2 } (\texttt{take } ((\texttt{length } \texttt{xs}) - \texttt{n}) \texttt{ xs})
\]

--- or ---

\[
\text{shave } \texttt{n } \texttt{xs} = \texttt{take } ((\texttt{length } \texttt{xs}) - 2^*\texttt{n}) (\texttt{drop } \texttt{n } \texttt{xs})
\]