Today

- Quiz 2
- List functions
- Types

Assignments

- HW-3, R-3 out
Using *where* blocks for local bindings

```haskell
lendAmt amt bal = 
  if newBal < reserve then 0 
  else amt 
  where reserve = 100 
       newBal = bal - amt
```

- Sometimes easier to read
- Has a different semantics when used with patterns (more later)
- e.g., can't be nested like a *let* expression

Both *where* and *let* can be used to define nested functions:

```haskell
-- avg of squared difference to the mean
variance2 mean x1 x2 =
  let squareDiff x = (x - mean)^2
  in (squareDiff x1 + squareDiff x2) / 2
```

```haskell
-- avg of squared difference to the mean
variance2' mean x1 x2 =
  (squareDiff x1 + squareDiff x2) / 2
  where squareDiff x = (x - mean)^2
```
List Operations (review) and Functions (cont)

List concatenation (++)

- Returns an entirely new list
- Values in the second list are appended to the values of the first list

Prelude> [3, 1, 3] ++ [3, 7]
[3,1,3,3,7]

Prelude> [True, False] ++ []
[True, False]

List construction (:)

- List construction is also often called “cons”
- Creates entirely new list with 1 prepended to values of given list
- Entire list can be written as a sequence of cons operations

Prelude> 1 : [2, 3]
[1,2,3]

Prelude> 1 : 2 : 3 : []
[1,2,3]

- Is cons (:) right or left associative?
  - Right associative!
  - 1 : (2 : (3 : []))
  - 1 : 2 is a type error since second operand is not a list!
**List head (aka “car”) gives first value of a list**

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

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

**List tail (aka “cdr”) gives list minus head value**

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

Prelude> tail [1]
[]

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

**List null checks for the empty list**

Prelude> null [4, 1, 5, 3]
False

Prelude> null []
True
More Haskell List Functions

List `length` gives number of elements in a list

```
Prelude> length [1..5]
5

Prelude> length []
0
```

List `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
```

List `reverse` gives list reversed

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

Prelude> reverse [1]
[1]

Prelude> reverse []
[]
```
**List** \( \text{take } n \) **gives first** \( n \) **elements as sublist**

```haskell
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]
[]
```

**List** \( \text{drop } n \) **gives list minus first** \( n \) **elements**

```haskell
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]
[]
```
List replicate \( n \) \( v \) returns a list of \( n \) values \( v \)

Prelude> replicate 5 1
[1,1,1,1,1]

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

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

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

List 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)]

List 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'
    ...

**List notElem x is false if x in list**

**Higher-order function filter removes elements of a list**

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

- We’ll talk about more functions on lists later ...
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 t) => t
```

- 49 has type t such that t 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’])
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

**Class constraints**

**For this type ...**

```
Prelude> :type 49
49 :: (Num t) => t
```

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