Today

- Quiz 7
- Haskell Types (cont)

Assignments

- HW 7 and Revised Proj Prop due
- HW 8 out
Let and where

We can define **local bindings** within functions

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

- similar to ghci let, but ghci let does not have an in clause
- let <bindings> in <expression> is itself an expression
- Can sometimes be more efficient (only evaluate expression once ...)

A let expression can be used in any subexpression:

```
Prelude> 3 + (let x=4 in x)
7

Prelude> 2 + (let x=3 in (let y=4 in x+y))
9
```

Note that parens are not needed above
Example where \texttt{let} is more efficient

\begin{verbatim}
checkVal x ys =
  if x == maximum ys
    then "x is max"
  else if x > maximum ys
    then "x is too high"
    else "x is not too high"
\end{verbatim}

Q: What is “inefficient” here?

\begin{itemize}
  \item we're calling \texttt{maximum} twice
  \item \texttt{maximum} needs to check all elements in \texttt{ys}
\end{itemize}

Q: How can we use \texttt{let} to make this more efficient?

\begin{verbatim}
checkVal x ys =
  let m = maximum ys
  in if x == m
    then "x is max"
    else if x > m
      then "x is too high"
      else "x is not too high"
\end{verbatim}

We’ll talk more about using \texttt{let} and recursion later ...
An alternative approach using where blocks

```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

-- avg of squared difference to the mean
variance2' mean x1 x2 =
  (squareDiff x1 + squareDiff x2) / 2
  where squareDiff x = (x - mean)^2
```
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'])
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 ...

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