Today

- Let and Where
- More on types

Assignments

- R-7, HW-7 due Tues
- Proj-1 due Today

Announcements

- Quiz on Tuesday
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:

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

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

Note that parens are not needed above
Example where `let` is more efficient

```haskell
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"
```

Q: What is “inefficient” here?

- we’re calling maximum twice
- maximum needs to check all elements in ys

Q: How can we use `let` to make this more efficient?

```haskell
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"
```

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

\begin{verbatim}
lendAmt amt bal = if newBal < reserve then 0 else amt
where reserve = 100
    newBal = bal - amt
\end{verbatim}

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

Both \texttt{where} and \texttt{let} can be used to define nested functions:

\begin{verbatim}
-- 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
\end{verbatim}
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
Function types

Functions have types (either given or inferred)

Prelude> not True
False

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

• The \(\rightarrow\) is read as “\(\text{to}\)” or “\(\text{returns}\)”

“not has the type \(\text{Bool} \text{ to} \text{ Bool}\)”

“not takes a \(\text{Bool}\) and returns a \(\text{Bool}\)”

Another example

Prelude> succ 6
7

Prelude> :type succ
succ :: (Enum a) => a -> a

• Here we have a class constraint

“for all \(\text{Enum}\) types \(a\), \(\text{succ}\) has the type \(a \text{ to} a\)”
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
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