This week:

- More basics
- Algebraic datatypes
- Polymorphism
- List functions
- List comprehensions
- Type synonyms
- Introduction to input/output (I/O)
- Compiling standalone programs
let and where (1)

- let:

```haskell
factorial :: Int -> Int
factorial n = 
  let iter n r = 
    if n == 0 then r 
    else iter (n-1) (n*r) 
  in 
    iter n 1
```
let and where (2)

- where:

```haskell
factorial :: Int -> Int
factorial n = iter n n 1
where
  iter n r =
    if n == 0 then r
    else iter (n-1) (n*r)
```
let and where (3)

- where (nicer):

```haskell
factorial :: Int -> Int
factorial n = iter n 1
  where
    iter 0 r = r
    iter n r = iter (n-1) (n*r)
```

```haskell
```
Lambda ($\lambda$) expressions

- Used to create anonymous functions

\[ \langle \text{pattern}\rangle \rightarrow \text{<expr>} \]

- Usually just e.g.

\[ \backslash x \rightarrow 2 \times x \]

\[ \backslash x \ y \rightarrow x + y \]

- Pattern example:

\[
\text{map } \left( \backslash (x, y) \rightarrow x + y \right)
\]

\[
\left[ (1, 2), (4, 1), (-3, 20) \right]
\]

\[ \Rightarrow [3, 5, 17] \]
Operator slices

- Instead of writing \( x \rightarrow x + 1 \)
- you can just write: 
  \((+1)\)
- Similarly, instead of writing \( x \rightarrow 2 \times x \)
- you can write: 
  \((2\times)\)
- Example:
  \(\text{map } (2\times) \ [1..5] \rightarrow [2,4,6,8,10]\)
case expressions (1)

- Used for pattern matches within expressions
- Syntax:
  ```
  case <expr> of
    <pattern1> -> <expr1>
    <pattern2> -> <expr2>
    ...
  ```
  - If want a default, use `_` (wildcard) as last pattern
  - `_` matches anything and throws the value away
case expressions (2)

- Example:
  
  ```haskell
  zeros :: [Int] -> [Int]
  zeros lst =
    case lst of
      (_ : rest) -> 0 : zeros rest
      [] -> []
  
  (Not terribly useful)
  
  Could also use pattern matching on function itself
Often want to define own data types to express the structure of some kind of data

Often the data can be in one of several alternative forms

Create an algebraic datatype for this

Many already provided in standard library

- AKA the Prelude
Example:

```haskell
data MaybeInt = NoInt | AnInt Int
let (x, y, z) = (NoInt, AnInt 2, AnInt 5)
```

- N.B. type names and data constructor names must start with capital letter!
- Type of `(x, y, z)`?
- `(MaybeInt, MaybeInt, MaybeInt)`
Algebraic datatypes (3)

- N.B. Can't define new datatypes in ghci
- Best to put into file and load using :l file.hs
- Might want to have a more general type than MaybeInt
  - the Maybe concept works just as well for any type
  - expresses concept of "not sure if will have anything, but if we do it'll be of this type"
- Don't want to have to define MaybeInt, MaybeFloat, MaybeString...
- All have same structure
Polymorphism (1)

- Data types can be parameterized over other types
- So for Maybe example we have (built-in):

```haskell
data Maybe a = Nothing | Just a
```

- Here `a` is a type variable
- Written with an initial lower-case letter
Polymorphism (2)

- **Types:**
  - Nothing :: Maybe a
  - Just 10 :: Maybe Int
  - Just "hi there!" :: Maybe String
  - Just :: a -> Maybe a

- Parameterized type constructors are also functions!

map Just [1..5]
→ [Just 1, Just 2, ..., Just 5]
Example

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

- Works for any list
More pattern matching

- Pattern matching on algebraic data types:
  
  ```haskell```
  foo :: Maybe Int -> Int
  foo Nothing = 0
  foo (Just x) = 1 + x

  bar :: Maybe (Maybe String) -> String
  bar Nothing = "None"
  bar (Just Nothing) = "Sorta"
  bar (Just (Just x)) = "Yes: " ++ x
  -- N.B. ++ concatenates lists
  ```haskell```
- Lists behave as if they were defined like this:
  ```haskell
  -- WARNING: Bogus pseudo-Haskell:
  data [a] = [] | a : [a]
  ```
- Note that `(:)` is a data constructor just like `Just`
- Pattern matching on lists:
  ```haskell
  head :: [a] -> Maybe a
  head (x : _) = Just x
  head [] = Nothing
  ```
- N.B. the parentheses are important!
As-patterns (@-patterns)

- You can assign a name to a pattern while also matching its parts
- Example:

```haskell
foo :: Maybe Int -> Maybe Int
foo x@(Just y) = x
foo Nothing = Nothing
```

- This looks useless now, but becomes useful when patterns get more complicated
List functions and the Prelude

- The Haskell Prelude is where the most basic functions are defined
- Always available to the programmer
- Includes many useful list functions
- Often fairly obvious what they do from the type signature
Useful list functions

- Examples:

  (+++) :: [a] -> [a] -> [a] -- list concat

  map :: (a -> b) -> [a] -> [b]

  filter :: (a -> Bool) -> [a] -> [a]

  head :: [a] -> a -- not like one we defined

  foldr :: (a -> b -> b) -> b -> [a] -> b

  repeat :: a -> [a]

  cycle :: [a] -> [a]
map

- `map f lst` applies `f` to each element of `lst`, returning the results

```haskell
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = f x : map f xs
map _ [] = []

map (/2) [1..3] \rightarrow [0.5, 1.0, 1.5]
```
foldr ("fold right")

- `foldr op z [x1, x2, ... xn]` reduces the list by computing `x1 `op` (x2 `op` ... (xn `op` z))`.
- Definition left as "exercise for student".

  ```haskell
  sum :: [Int] -> Int
  sum = foldr (+) 0
  ```

- Pop quiz: what is `foldr (:) []`?
More list functions (1)

- `concat :: [[a]] -> [a]`
- `take :: Int -> [a] -> [a]`
- `drop :: Int -> [a] -> [a]`
- `elem :: a -> [a] -> Bool`
  -- usually written as
  -- 5 `elem` [1..10]
- `zip :: [a] -> [b] -> [(a, b)]`
More list functions (2)

- Many more list functions in Prelude
- Use Prelude functions instead of reimplementing them yourself
  - read Prelude docs (linked from web pages)
List comprehensions (1)

- List comprehensions are a convenient way to create lists with particular properties

\[
fibs :: [\text{Integer}] \\
fibs = 0:1:[x+y|(x,y) \leftarrow \text{zip fibs (tail fibs)}]
\]

- Infinite list of fibonacci numbers

- To get first 20, do

\[
\text{take 20 fibs}
\]
List comprehensions (2)

- General structure:
  \[
  [<\text{expr}> \mid \text{pattern} \leftarrow \text{source} \ldots, \\
  \quad \text{filter} \ldots]
  \]

- Examples:
  \[
  [x \mid x \leftarrow [1..1000], x \ `\text{mod}` \ 2 == 1]
  \]
  \[
  [(x, y) \mid x \leftarrow [1..10], y \leftarrow [1..10], \\
  x + y == 10]
  \]
Type synonyms

- Can create a synonym for a type
- Compiler can't always figure out the right name to use (e.g. in ghci), but it tries

Examples:

type String = [Char]  -- in the Prelude
type Label = String
type Point = (Double, Double)
**Introduction to I/O (1)**

- Input/output is odd in Haskell
- Can't have side effects!
- Input/output actions are values of type `IO a`, where `a` is the type of the action's result
- Actions with no useful result have type `IO ()`
  - `()` is the sole instance of the unit type
Introduction to I/O (2)

- Examples:
  - `putStrLn :: String -> IO ()`
  - `putStrLn :: String -> IO ()`
  - `getLine :: IO String`
  - `print :: a -> IO ()`

- Our first encounter with dreaded Monads
  - Much more to say about this in future

- Entire program is a computation of type `IO ()`
Compiling standalone programs

- Create a main function with type `IO ()`
- Compile the program with:
  \% ghc -o programe filename.hs
- Run the program:
  \% programe
- Hit ctrl-C if the program doesn't terminate
Next week

- Much more on I/O
- Type classes