Haskell Hodgepodge

- Lists
- Universal polymorphism (aka "generics")
- Function notations: unnamed, infix, prefix, sections, parenthesis avoidance ($)
- Guards
- `map`, `filter`, `all`

Lists

"List" is a *type constructor".

For any element type `a`, `[a]` is the type of lists with elements in `a`

```
                x ∈ a   l ∈ [a]
(1)----------   (2)----------------
    [] ∈ [a]        (:) x l ∈ [a]
```

E.g. if `a` is `Int`:

```
1.  [] ∈ [Int]                          by rule (1).
2.  (:) 17 [] ∈ [Int]                   by rule (2) using line 1.
3.  (:) 3 ((:) 17 []) ∈ [Int]           by rule (2) using 2.
4.  (:) 5 ((:) 3 ((:) 17 [])) ∈ [Int]   by rule (2) using line 3.
```

Notation: Haskell will display 4. as `[5,3,17]` but this is just *notation*, or *surface syntax*.

Aside: infix vs prefix notations for functions

```
f x y = x ++ y
ex = f "a" "b" ++ "c"
--    ^^^^^^^^^----------> "prefix" application of f to "a" and "b"
--    ^^^^^^^^^^^^^^^^---> "infix" application of ++ to (f "a" "b") and "c"
```

Haskell has notation for going back and forth between prefix and infix:

- If o is infix, then (o) is prefix
- If o is prefix, then `` `o` `` is infix

```
eg = (++) ("a" `f` "b") "c"
```

Note: `(:)` has parens above because `:` is a prefix operator.

The following are the same in Haskell:

(1) `5 : 3 : 17 : []`, (2) `5 : (3 : (17 : []))`, and (3) `[5,3,17]`

Patterns over lists

```
f :: [Int] -> Bool
f [] = True
f (2 : l) = (length l == 2)
f [1, 2, x] = (x == 17)
f _ = False
```

E.g. draw trees for the pattern matching for `f [1, 2, 42]`.

Universal polymorphism

```
tail :: [?] -> [?]
tail [] = []
tail (x : l) = l
```

This function works for any list, regardless of the element type `?`.
We can use a *type variable*, e.g. `a` to stand for the element type.

```
tail :: [a] -> [a]
```

means that for any type `a`, tail can take a list with elements in `a` and produce another list with elements in `a`.

Basic pattern-matching and recursion for lists.

A list in *[a]* is either

- *[]* or
- *(x : l)* for some $x \in a$ and $l \in [a]$.

```
listize :: [a] -> [[a]]
listize [] = []
listize (x : l) = [x] : listize l
```

```haskell
append :: [a] -> a -> [a]
append  [] y = [y]
append (x : l) y = x : append l y
```

See Haskell file for further examples.

Sections: giving an argument "in advance"

An example is sufficient.

```haskell
(+++) :: Int -> Int -> Int
x +++ y = x + 2*y
```

```
ghci> :type (1 +++)
(1 +++) :: Int -> Int
ghci> :type (+++ 1)
(+++ 1) :: Int -> Int
ghci> (1 +++) 2
5
ghci> (+++ 1) 2
4
```

So, eg: `map (+ 1) [1,2,3] = [2,3,4]`

Unnamed functions

A *lambda expression* is a way of writing a function without a definition.

If

```
f x = 2*x + 17
```

then `(\x -> 2*x + 17)` and `f` are the same as functions. Computing:

```
(\x -> 2*x + 17) 2  =  2*2 +17  =  21  =  f 2
```

Can use a lambda-expression anywhere a function is expected.

E.g. the value of `map (\x -> x+1) [1,2,3]` is `[2,3,4].`

Optional convenience: $

`f $ x` means the same thing as `f x`, i.e. apply the function `f` to `x`.

Why bother? Because, e.g. Haskell takes `f $ g x` to mean `f (g x)`.
It's just a way of avoiding writing parentheses.

```
BT a = Leaf a | Node (BT a) (BT a)

left(Node l _) = l
right(Node _ r) = r

-- Follow left branches three times, assuming it's possible
lll :: BT a -> BT
lll t = l $ l $ l  t  -- l (l (l t))
```

Guards

Enhancement for pattern-matching equations.

A left-hand-side can be followed by `| e` where `e` is a boolean expression.

The equation is used only if the pattern matches *and* `e` is true.

```
absVal n | n < 0 = -n
absVal n = n
```

```
-- insert into a sorted list
-- insert 3 [1,2,3,4] = [1,2,3,3,4]
insert :: Int -> [Int] -> [Int]
insert x [] =
  [x]
insert x (y : l) | x <= y =
  x : y : l
insert x (y : l) =
  y : (insert x l)
```