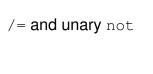
#### str1 ++ str2

Any two lists can be concatenated this way. Strings are just lists of characters.



if bool then expr1 else expr2

The else clause is required, to ensure an interesting result for the whole  ${\tt if}$  expression.



:t expr shows a value's type in ghci.

:set +t makes it permanent for that session.

The optional type declaration and the function declaration.

module MyMath where
 my\_max :: Integer -> Integer -> Integer
 my\_max x y = if x > y then x else y

- Multiple function definitions using pattern-matching

parameters. - Guards.

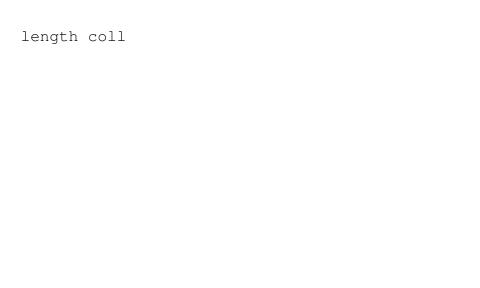
#### Delay the equals sign, splitting up the argument lists.

otherwise is an alias for true, often used as the last guard.

Tuples use round parens, lists use brackets.

### You can do it explicitly with parens and parameters, or with the dot notation:

composedFunc arg = secondToApply (firstToApply arg)
composedFunc = secondToApply . firstToApply



```
[start ..] --infinite list
[start .. end] --default increment of 1
[start, second .. end] --uses second to show increment
```

[expr | genOrFilter1, ..., genOrFilterN]

#### Filters are boolean expressions.

Generator form:

bindingForm <- collection</pre>

\param1 ... paramN -> body

# You can append an indented where clause.

Just don't give a function all its arguments. No special syntax

is required.

data	NewType	=	Val1	I	Val2			ValN	

```
type NewType = OldType
type NewType = (Type1, ..., TypeN)
type NewType = [SomeType]
```

## Add deriving (Show) to the end of the constructor. $\mathbf{k}$

# They are interpreted as type variables.

A collection of function signatures that any instance of the class (a **type** not an object) must support.

```
let var1 = expr1
```

in result-expr

var2 = expr2

... a way of composing functions.

... sequential execution, control structures, managing i/o, the

Maybe monad.