```
str1 ++ str2
```

Lists of characters.

`/=` and unary `not`

```
if bool then expr1 else expr2
```

The else clause is required, to ensure an interesting result for the whole `if` expression.

... strict.

`: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.

```
func args
    | boolean1 = ...
    | boolean2 = ...
```

`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
```

```
length coll
```

```
[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
```

```
\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 | Val2 | ... | ValN
```

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

Add `deriving (Show)` to the end of the constructor.
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
    ...
    var2 = expr2
in result-expr
```

... a way of composing functions.

... sequential execution, control structures, managing i/o, the `Maybe` monad.