Syntax

Comments

```
-- single line comment

{- multiline
   comment
-}
```

Basic types

```
True  : Bool
False : Bool

13    : number  -- Int or Float depending on usage
4.2   : Float

"hello world" : String
'a'   : Char
```

String manipulation

```
"hello" ++ " world" -- Concatenation
"age: " ++ (toString 42) -- Concatenation with numbers
```

Functions

```
add a b = a + b -- Definition
add 3 5 -- Example of use, returns 8
anotherAdd = (\a b -> a + b) -- Lambda / anonymous function
anotherAdd 3 5 -- Returns 8
```

Lists

```
["elem1", "elem2"]
1 :: [2,3,4] -- Equals [1, 2, 3, 4]
List.map (\a -> a * 2) [1, 2, 3, 4] -- Equals [2, 4, 6, 8]
```

Tuples

```
("text", 42) -- Tuple with two values
(34, 23, "text") -- Tuple with 3 values
```

More details about tuples

Conditions

```
myAge = 19
if myAge >= 18 then "overage" else "underage"
-- Returns "overage"
```

Records

```
myUser = { username = "Marcus", password = "Secr3T" }
myUser.username -- Returns "Marcus"
```

Types

```
-- Type alias
type alias User = { username = String, password = String }

-- Gives you a "constructor" function 'User'
User "Marcus" "Secr3T" == { username = "Marcus", password = "Secr3T" }


-- Union type
type RemoteString = NotLoaded | Loading | Loaded String | OnError Int

{- For null values, you can use the Maybe union type
   that is either "Just something" or "Nothing":
-}
Just "value" : Maybe.Maybe String
Nothing : Maybe.Maybe a
```

More informations about union types

Cases

Match values against patterns

```
case myListOfString of
  [] -> "empty" -- Handles empty list
  head :: others -> head
  -- Store head of the list in head and the rest of the list in the List others

case myStringMaybe of
  Just value -> value
  Nothing -> "empty string"

case myNumber of
  0 -> "the number is 0"
  1 -> "the number is 1"
  _ -> "the number is neither 0 nor 1"
```

Let Expressions

`let` these values be defined `in` this specific expression.

```
let
  days =
    2
  seconds =
    numberOfDays * 24 * 60 * 60
in
  (toString seconds) " seconds in " ++ (toString days) ++ " days"
```

Allows to split complex expressions into smaller definitions to ease the read. Use when needed but be careful, sometimes it is better to extract code into functions!

Modules

```
module MyModule exposing (..)

import List exposing (..) -- import module and expose everything
import List exposing ( map, foldl ) -- exposes only map and foldl
```

Qualified imports are preferred. Module names must match their file name, so module `Parser.Utils` needs to be in file `Parser/Utils.elm`.

Type Annotations

You can learn more about type annotations here.