Elm

Elm is a functional reactive programming language that compiles to (client-side) JavaScript. Elm is statically typed, meaning that the compiler catches most errors immediately and provides a clear and understandable error message. Elm is great for designing user interfaces and games for the web.

```
-- Single line comments start with two dashes.
{- Multiline comments can be enclosed in a block like this.
{- They can be nested. -}
-}

{-- The Basics --}

-- Arithmetic
1 + 1 -- 2
8 - 1 -- 7
10 * 2 -- 20

-- Every number literal without a decimal point can be either an Int or a Float.
33 / 2 -- 16.5 with floating point division
33 // 2 -- 16 with integer division

-- Exponents
5 ^ 2 -- 25

-- Booleans
not True -- False
not False -- True
1 == 1 -- True
1 /= 1 -- False
1 < 10 -- True

-- Strings and characters
"This is a string because it uses double quotes."
'a' -- characters in single quotes

-- Strings can be appended.
"Hello " ++ "world!" -- "Hello world!"

{-- Lists, Tuples, and Records --}

-- Every element in a list must have the same type.
["the", "quick", "brown", "fox"]
[1, 2, 3, 4, 5]
-- The second example can also be written with two dots.
[1..5]

-- Append lists just like strings.
[1..5] ++ [6..10] == [1..10] -- True

-- To add one item, use "cons".
0 :: [1..5] -- [0, 1, 2, 3, 4, 5]

-- The head and tail of a list are returned as a Maybe. Instead of checking
-- every value to see if it's null, you deal with missing values explicitly.
List.head [1..5] -- Just 1
List.tail [1..5] -- Just [2, 3, 4, 5]
List.head [] -- Nothing
-- List.functionName means the function lives in the List module.

-- Every element in a tuple can be a different type, but a tuple has a
-- fixed length.
("elm", 42)

-- Access the elements of a pair with the first and second functions.
-- (This is a shortcut; we'll come to the "real way" in a bit.)
fst ("elm", 42) -- "elm"
snd ("elm", 42) -- 42

-- The empty tuple, or "unit", is sometimes used as a placeholder.
-- It is the only value of its type, also called "Unit".
()

-- Records are like tuples but the fields have names. The order of fields
-- doesn't matter. Notice that record values use equals signs, not colons.
{ x = 3, y = 7 }

-- Access a field with a dot and the field name.
{ x = 3, y = 7 }.x -- 3

-- Or with an accessor fuction, which is a dot and the field name on its own.
.y { x = 3, y = 7 } -- 7

-- Update the fields of a record. (It must have the fields already.)
{ person |
  name = "George" }

-- Update multiple fields at once, using the current values.
{ particle |
  position = particle.position + particle.velocity,
  velocity = particle.velocity + particle.acceleration }

{-- Control Flow --}

-- If statements always have an else, and the branches must be the same type.
if powerLevel > 9000 then
  "WHOA!"
else
  "meh"

-- If statements can be chained.
if n < 0 then
  "n is negative"
else if n > 0 then
  "n is positive"
else
  "n is zero"
```

```
-- Use case statements to pattern match on different possibilities.
case aList of
  [] -> "matches the empty list"
  [x]-> "matches a list of exactly one item, " ++ toString x
  x::xs -> "matches a list of at least one item whose head is " ++ toString x
-- Pattern matches go in order. If we put [x] last, it would never match because
-- x::xs also matches (xs would be the empty list). Matches do not "fall through".
-- The compiler will alert you to missing or extra cases.

-- Pattern match on a Maybe.
case List.head aList of
  Just x -> "The head is " ++ toString x
  Nothing -> "The list was empty."

{-- Functions --}

-- Elm's syntax for functions is very minimal, relying mostly on whitespace
-- rather than parentheses and curly brackets. There is no "return" keyword.

-- Define a function with its name, arguments, an equals sign, and the body.
multiply a b =
  a * b

-- Apply (call) a function by passing it arguments (no commas necessary).
multiply 7 6 -- 42

-- Partially apply a function by passing only some of its arguments.
-- Then give that function a new name.
double =
  multiply 2

-- Constants are similar, except there are no arguments.
answer =
  42

-- Pass functions as arguments to other functions.
List.map double [1..4] -- [2, 4, 6, 8]

-- Or write an anonymous function.
List.map (\a -> a * 2) [1..4] -- [2, 4, 6, 8]

-- You can pattern match in function definitions when there's only one case.
-- This function takes one tuple rather than two arguments.
area (width, height) =
  width * height

area (6, 7) -- 42

-- Use curly brackets to pattern match record field names.
-- Use let to define intermediate values.
volume {width, height, depth} =
  let
    area = width * height
  in
```

```
    area * depth

volume { width = 3, height = 2, depth = 7 } -- 42

-- Functions can be recursive.
fib n =
  if n < 2 then
    1
  else
    fib (n - 1) + fib (n - 2)

List.map fib [0..8] -- [1, 1, 2, 3, 5, 8, 13, 21, 34]

-- Another recursive function (use List.length in real code).
listLength aList =
  case aList of
    [] -> 0
    x::xs -> 1 + listLength xs

-- Function calls happen before any infix operator. Parens indicate precedence.
cos (degrees 30) ^ 2 + sin (degrees 30) ^ 2 -- 1
-- First degrees is applied to 30, then the result is passed to the trig
-- functions, which is then squared, and the addition happens last.

{-- Types and Type Annotations --}

-- The compiler will infer the type of every value in your program.
-- Types are always uppercase. Read x : T as "x has type T".
-- Some common types, which you might see in Elm's REPL.
5 : Int
6.7 : Float
"hello" : String
True : Bool

-- Functions have types too. Read -> as "goes to". Think of the rightmost type
-- as the type of the return value, and the others as arguments.
not : Bool -> Bool
round : Float -> Int

-- When you define a value, it's good practice to write its type above it.
-- The annotation is a form of documentation, which is verified by the compiler.
double : Int -> Int
double x = x * 2

-- Function arguments are passed in parentheses.
-- Lowercase types are type variables: they can be any type, as long as each
-- call is consistent.
List.map : (a -> b) -> List a -> List b
-- "List dot map has type a-goes-to-b, goes to list of a, goes to list of b."

-- There are three special lowercase types: number, comparable, and appendable.
-- Numbers allow you to use arithmetic on Ints and Floats.
-- Comparable allows you to order numbers and strings, like a < b.
-- Appendable things can be combined with a ++ b.
```

```
{-- Type Aliases and Union Types --}

-- When you write a record or tuple, its type already exists.
-- (Notice that record types use colon and record values use equals.)
origin : { x : Float, y : Float, z : Float }
origin =
  { x = 0, y = 0, z = 0 }

-- You can give existing types a nice name with a type alias.
type alias Point3D =
  { x : Float, y : Float, z : Float }

-- If you alias a record, you can use the name as a constructor function.
otherOrigin : Point3D
otherOrigin =
  Point3D 0 0 0

-- But it's still the same type, so you can equate them.
origin == otherOrigin -- True

-- By contrast, defining a union type creates a type that didn't exist before.
-- A union type is so called because it can be one of many possibilities.
-- Each of the possibilities is represented as a "tag".
type Direction =
  North | South | East | West

-- Tags can carry other values of known type. This can work recursively.
type IntTree =
  Leaf | Node Int IntTree IntTree
-- "Leaf" and "Node" are the tags. Everything following a tag is a type.

-- Tags can be used as values or functions.
root : IntTree
root =
  Node 7 Leaf Leaf

-- Union types (and type aliases) can use type variables.
type Tree a =
  Leaf | Node a (Tree a) (Tree a)
-- "The type tree-of-a is a leaf, or a node of a, tree-of-a, and tree-of-a."

-- Pattern match union tags. The uppercase tags will be matched exactly. The
-- lowercase variables will match anything. Underscore also matches anything,
-- but signifies that you aren't using it.
leftmostElement : Tree a -> Maybe a
leftmostElement tree =
  case tree of
    Leaf -> Nothing
    Node x Leaf _ -> Just x
    Node _ subtree _ -> leftmostElement subtree

-- That's pretty much it for the language itself. Now let's see how to organize
-- and run your code.
```

```
{-- Modules and Imports --}

-- The core libraries are organized into modules, as are any third-party
-- libraries you may use. For large projects, you can define your own modules.

-- Put this at the top of the file. If omitted, you're in Main.
module Name where

-- By default, everything is exported. You can specify exports explicity.
module Name (MyType, myValue) where

-- One common pattern is to export a union type but not its tags. This is known
-- as an "opaque type", and is frequently used in libraries.

-- Import code from other modules to use it in this one.
-- Places Dict in scope, so you can call Dict.insert.
import Dict

-- Imports the Dict module and the Dict type, so your annotations don't have to
-- say Dict.Dict. You can still use Dict.insert.
import Dict exposing (Dict)

-- Rename an import.
import Graphics.Collage as C

{-- Ports --}

-- A port indicates that you will be communicating with the outside world.
-- Ports are only allowed in the Main module.

-- An incoming port is just a type signature.
port clientID : Int

-- An outgoing port has a defintion.
port clientOrders : List String
port clientOrders = ["Books", "Groceries", "Furniture"]

-- We won't go into the details, but you set up callbacks in JavaScript to send
-- on incoming ports and receive on outgoing ports.

{-- Command Line Tools --}

-- Compile a file.
$ elm make MyFile.elm

-- The first time you do this, Elm will install the core libraries and create
-- elm-package.json, where information about your project is kept.

-- The reactor is a server that compiles and runs your files.
-- Click the wrench next to file names to enter the time-travelling debugger!
$ elm reactor

-- Experiment with simple expressions in a Read-Eval-Print Loop.
```

```
$ elm repl

-- Packages are identified by GitHub username and repo name.
-- Install a new package, and record it in elm-package.json.
$ elm package install evancz/elm-html

-- See what changed between versions of a package.
$ elm package diff evancz/elm-html 3.0.0 4.0.2
-- Elm's package manager enforces semantic versioning, so minor version bumps
-- will never break your build!
```

The Elm language is surprisingly small. You can now look through almost any Elm source code and have a rough idea of what is going on. However, the possibilties for error-resistant and easy-to-refactor code are endless!

Here are some useful resources.

- The Elm website. Includes:
- Links to the installers
- Documentation guides, including the syntax reference
- Lots of helpful examples
- Documentation for Elm's core libraries. Take note of:
- Basics, which is imported by default
- Maybe and its cousin Result, commonly used for missing values or error handling
- Data structures like List, Array, Dict, and Set
- JSON encoding and decoding
- The Elm Architecture. An essay by Elm's creator with examples on how to organize code into components.
- The Elm mailing list. Everyone is friendly and helpful.
- Scope in Elm and How to Read a Type Annotation. More community guides on the basics of Elm, written for JavaScript developers.

Go out and write some Elm!