A Concise Introduction to Elm

General

Elm is a functional *reactive* language. That is, rather than "reading" inputs from some point in the program, input events just "happen," and the program reacts to them. This is the way GUI programs are usually written, using callbacks, but Elm is designed to do much the same thing without the complexity of callbacks.

Elm compiles into JavaScript, so Elm programs are normally run in a browser, and make use of a lot of functions that compile into HTML tags.

Running programs

Online

To run in an online IDE, go to http://elm-lang.org/try. This is a good way to write small programs.

Installing Elm

Use the installer from http://elm-lang.org/install.

Local REPL

To try out expressions in the REPL, type `elm repl` or `elm-repl` at the command line. Exit with `:exit`.

Rapid feedback

In the root directory of your project, execute `elm-reactor`. Then in a browser, go to http://localhost:8000. Click on your Elm file in the displayed page to run it.

You can make changes in your favorite editor (Sublime Text has good syntax coloring for Elm), save the file, and click the refresh button in your browser.

Downside: The Elm reactor uses a default CSS file, not one you supply.

Compiling to JavaScript

Create a directory for your project and navigate into it. By convention, create a `Main.elm` file in this directory.

Compile with `elm-make Main.elm --output index.html`. Then open the result in a browser.

You can also do `elm-make Main.elm --output main.js`, then link to the resulting JavaScript in an HTML page.

To compile `hello.elm` to a `hello.js` file, execute the following in the desired directory:

```
elm package install
elm package install evancz/elm-html
elm package install evancz/start-app
elm make hello.elm --output hello.html
elm make hello.elm --output hello.js
```

Then

Imports

Many functions are in modules that must be imported, such as the `String` and `List` modules.

- `import String` imports the `String` module, and allows *qualified* references to functions in the module, such as `String.toUpper`.
- `import String exposing (toUpper, toLower)` imports the `String` module, and allows *unqualified* references to `toUpper` and `toLower`, and qualified references to other functions in the module.
- `import String exposing (..)` imports the `String` module and allows *unqualified* references to all functions in the module.

Comments

- `--` begins a single line comment.
- `{-` and `-}` enclose a multiline comment; these can be nested.

Types

Elm has:

- Strings: `"hello"`. Strings are *not* lists of characters.
- Chars: `'a'`
- Functions: `isNegative n = n < 0`
- Lists (all elements must have the same type): `["one", "two", "three"]`
- Tuples (fixed number of values, any mix of types): `("Dave", True)`
- Records (key-value pairs): `{ x = 0, y = 10 }`
- Named functions: `avg x y = (x + y) / 2`
- Anonymous functions: `(\ x y -> (x + y) / 2)`
- Aliases: `type alias Point = { x = 0, y = 0 }`

Operations

Operations on numbers

- Arithmetic expressions: `+`, `-`, `*`, `/` (result is always a float), `//` (integer division), `%` (mod), parentheses
- Comparators: `<`, `<=`, `==`, `!=`, `>=`, `>`

Operations on booleans

- `&&`, `||`, `not`

Operations on strings

- `++` is string concatenation

Operations on lists

```
import List exposing (..)
```

`(::) : a -> List a -> List a`
"Cons" (add) an element to a list.

`head : List a -> a`
Return the first element of a list.

`tail : List a -> List a`
Return the remainder of a list after the head.

`member : a -> List a -> Bool`
Tests whether a value is in a list.

`take : Int -> List a -> List a`
Returns the first n elements of a list.

`scanl : (a -> b -> b) -> b -> List a -> List b`
Reduce a list from the left, building up all of the intermediate results into a list.

`scanl1 : (a -> b -> b) -> b -> List a -> List b`
Reduce a non-empty list from the left, building up all of the intermediate results into a list.

`map2 : (a -> b -> result) -> List a -> List b -> List result`

`drop : Int -> List a -> List a`
: Returns the list with the first n elements removed.

`isEmpty : List a -> Bool`
: Tests if a list is empty.

`length : List a -> Int`
: Returns the length of a list.

`reverse : List a -> List a`
: Reverses a list.

`append : List a -> List a -> List a`
: Appends two lists.

`concat : List (List a) -> List a`
: Combine a list of lists into a single list.

`intersperse : a -> List a -> List a`
: Put an element between all elements of a list.

`map : (a -> b) -> List a -> List b`
: Applies a function to each element of a list, returning a list of results.

`filter : (a -> Bool) -> List a -> List a`
: Applies a predicate to each element of a list, retaining those that satisfy the predicate.

`foldl : (a -> b -> b) -> b -> List a -> b`
: Reduce a list from the left. Specifically, takes a binary function `f a b -> b`, an initial value `b`, and a list of `a`, producing a single `b`.

`foldl1 : (a -> a -> a) -> List a -> a`
: Reduce a non-empty list from the left.

`foldr : (a -> b -> b) -> b -> List a -> b`
: Reduce a list from the right.

`foldr1 : (a -> b -> b) -> b -> List a -> b`
: Reduce a non-empty list from the right.

: Combine two lists, element-wise, with the given function. The functions `map3`, `map4`, and `map5` also exist.

`zip`
: Not currently in Elm; use `map2 (,)`.

`unzip : List (a, b) -> (List a, List b)`
: Given a list of tuples, returns a tuple of lists.

`partition : (a -> Bool) -> List a -> (List a, List a)`
: Returns a tuple of the elements that satisfy the predicate and those that fail the predicate.

`all : (a -> Bool) -> List a -> Bool`
: Tests whether all elements satisfy the predicate.

`any : (a -> Bool) -> List a -> Bool`
: Tests whether any element satisfies the predicate.

`sum : List number -> number`
: Returns the sum of the list elements.

`product : List number -> number`
: Returns the product of the list elements.

`maximum : List comparable -> Maybe comparable`
: Returns the largest value in the list.

`minimum : List comparable -> Maybe comparable`
: Returns the smallest value in the list.

`sort : List comparable -> List comparable`
: Sorts from lowest to highest.

`sortBy : (a -> comparable) -> List a -> List a`
: Sorts by a property of the list elements, such as a field of a record.

`sortWith : (a -> a -> Order) -> List a -> List a`
: Sorts according to the supplies function.

`filterMap : (a -> Maybe b) -> List a -> List b`
: Apply a function to a list and keep only the ones that succeed in returning a value.

`concatMap : (a -> List b) -> List a -> List b`
: Map a function onto a list and flatten the results ("flatmap" in some languages).

`indexedMap : (Int -> a -> b) -> List a -> List b`
: Same as map but the function is also applied to the index of each element (starting at zero).

Operations on tuples

`(,)`, `(,,)`, `(,,,)`, etc.
: Functions that construct tuples from arguments.

`fst : (a, b) -> a`
: Returns the first element.

`snd : (a, b) -> b`
: Returns the second element.

Pattern matching in `case` expressions can be used on tuples. Beyond that, Elm has very little support for tuples.

Operations on records

A record is like a Python dictionary or a Java HashMap

- `{ name = value, ..., name = value }` defines a record.
- Use `record.key` or `.key record` to access the fields of a record.
- http://elm-lang.org/docs/syntax#records has a number of operations without much explanation.
- To copy a record but with some fields different, use
  `{ old_record | key1 <- new_value_1, ..., key_n <- new_value_n }`

Conditional expressions

- `if condition then value_if_True else value_if_False`
- ```
  if | condition -> expression
     | condition -> expression
     ...
     | otherwise -> expression
  ```
- ```
  case expression of
    pattern -> expression
    ...
    pattern -> expression
  ```

Calling functions

As is usual in functional languages, functions are *curried*--they only take a single parameter. A function that appears to take several parameters actually takes only the first parameter, returning a function that takes the next parameter, etc., until a single result is obtained.

To call a function, give its name and its parameters, separated by spaces, for example,
```
List.map toUpper words
```

The result of a function can be *piped* to the next function, using the `|>` operator, for example,

```
message ++ "  "
   |> String.toUpper
   |> String.repeat times
   |> String.trimRight
   |> Html.text
```

If the argument to a function is a record, you can specify which fields must be present. Example:
```
rec = {x = 5, y = 7, z = 3}
maxxz {x, z} = if x > z then x else z
maxxz rec -- returns 5
```

In the REPL, a backslash (`\`) at the end of a line causes the next line to act as a continuation of the current line. Be sure not to have any whitespace after the backslash.

Signals and maps

Signals

The *reactive* part of Elm is its use of *signals*. A signal is a variable whose value varies according to external events, such as mouse motion or keyboard entries. Here are some of the supplied signals:

- `import Mouse exposing (..)`
  - `position : Signal ( Int, Int )`
  - `x : Signal Int`
  - `y : Signal Int`
  - `isDown : Signal Bool`
  - `clicks : Signal ()`
- `import Keyboard exposing (..)`
  - `Mouse.position : Signal (Int, Int)`
  - `arrows : Signal { x : Int, y : Int }`
  - `wasd : Signal { x : Int, y : Int }`
  - `enter`, `space`, `ctrl`, `shift`, `alt`, `meta` all have type `Signal Bool`
- `type alias KeyCode = Int`
  - `isDown : KeyCode -> Signal Bool`
  - `keysDown : Signal (Set KeyCode)`
    `presses : Signal KeyCode` -- most recent key pressed
- `import Time exposing (..)`
  - `fps : number -> Signal Time` (frames per second) will produce a signal the given number of times every second
  - `fpsWhen : number -> Signal Bool -> Signal Time`
    - Same as the `fps` function, but you can turn it on and off
  - `every : Time -> Signal Time` takes a time interval `t` and produces a signal updated every `t`
  - `delay : Time -> Signal a -> Signal a` delays a time signal
  - There a few additional `Time` functions
- `import Window exposing (..)`
  - `dimensions : Signal ( Int, Int )`
  - `width : Signal Int`
  - `height : Signal Int`
- `foldp : (a -> state -> state) -> state -> Signal a -> Signal state`
  - `foldp` folds signals "over time"; used to step the state of the computation forward

Maps

Variables in a purely functional language should be immutable. Signals are not immutable, therefore they are kept in "isolation" in a Signal *monad*. Values in a Signal can be used to call pure functions, but the result must be put immediately back into a Signal. This is the purpose of the various `map` functions.

- `Signal.map : (a -> b) -> Signal a -> Signal b`
  - This function was originally named `lift`. It has an argument structure similar to `List.map`, `(a -> b) -> List a -> List b`, and I suspect the function was renamed to take advantage of the familiarity of the latter.

Some pure functions may require more than one argument, so there are functions for calling pure functions with more than one Signal.

- `map2 : (a -> b -> result) -> Signal a -> Signal b -> Signal result`
- `map3 : (a -> b -> c -> result) -> Signal a -> Signal b -> Signal c -> Signal result`
- `map4 : (a -> b -> c -> d -> result) -> Signal a -> Signal b -> Signal c -> Signal d -> Signal result`
- `map5 : (a -> b -> c -> d -> e -> result) -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e -> Signal result`

There are several additional functions for working with and combining Signals.

- `merge : Signal a -> Signal a -> Signal a`
- `mergeMany : List (Signal a) -> Signal a`
- `keepIf : (a -> Bool) -> a -> Signal a -> Signal a` -- the first `Signal a` is a default value
- `keepWhen : Signal Bool -> a -> Signal a -> Signal a` -- the first `Signal a` is a default value
- dropRepeats : Signal a -> Signal a
- constant : a -> Signal a

In addition,

- `sampleOn : Signal a -> Signal b -> Signal b`
  - Samples from the second input every time an event occurs on the first input. I believe the purpose of this is to keep the program from having to deal with an infeasibly large number of signals.

Structure of an Elm program

GUI programs are usually written using the MVC (Model-View-Controller) pattern. Elm programs are written following a very similar pattern, as follows (http://elm-lang.org/guide/architecture#the-basic-pattern):

MODEL	UPDATE	VIEW

`type alias Model = { ... }`	`type Action = NoOp | Move Int Int | ...`
	
	`update : Action -> Model -> Model`
	`update action model =`
	`  case action of`
	`  NoOp -> ...`
	`  Move x y -> ...`
	`  ...`
		`view : Model -> Html`
		`view = ...`
		
		`main : Signal Element`
		`main = ...`

The `Model` is a record that describes the current state of the program.
An `Action` is a list of type names provided by the programmer, naming the types of action that can occur. Actions may have parameters.
The `update` method applies an `Action` to a `Model`, producing a new `Model`.
Note that, in the above template, the output is an HTML page.

Imports

The follow lists some of the libraries that may need to be imported, along with the methods found in each. As Elm is a rapidly evolving language, the URLs provided may link to outdated versions, which should in turn link to newer versions.

Keyboard 2.1.0

- `type alias KeyCode = Int`
- `arrows : Signal { x : Int, y : Int }`
- `wasd : Signal { x : Int, y : Int }`
- `enter`, `space`, `ctrl`, `shift`, `alt`, `meta`, `isDown` are all `: Signal Bool`
  - The meta key is the Windows key on Windows and the Command key on Mac.
- `keysDown : Signal (Set KeyCode)`
- `presses : Signal KeyCode`

```
main =  Signal.map show Keyboard.keysDown --shows keyCodes for currently pressed keys
```

← 37, ↑ 38, → 39, ↓ 40

Mouse 2.1.0

- `position : Signal (Int, Int)`
- `x : Signal Int`
- `x : Signal Int`
- `isDown : Signal Bool` (State of the left mouse button; apparently no way to check the right mouse button.)
- `clicks : Signal ()` (Event triggers on every mouse click.)

HTML

The syntax of an HTML tag is `<tag attributes> Contents </tag>`. In Elm this is represented as a function with two list arguments: `tag [attributes] [Contents]`. The attributes are of type `Attribute`, and there are assorted methods in `Html.Attributes` for creating these.

The following types can be displayed: `Element` , `Html`, `(Signal Element)`, `(Signal Html)`.

`show : a -> Element`
: Converts any type of value to a displayable Element. Strings and characters are shown with enclosing quote marks.

`text : String -> Svg.Svg`
: Turns a String into a graphical element that can be displayed.

Html.ul [] [li [] [text "Hello"], li [] [text "there"]]