Elm Lesson One

Welcome to an incredible journey to learn and use the "delightful" Elm computer language used to build web pages. This lesson assumes that the student has read the first sections of "Elm for Beginners" at least up to "The Characteristics of the Elm Language" and perhaps skimmed much of the rest of that document, skipping over the "Advanced Container Manipulation" section. This lesson doesn't assume any prior knowledge about computer languages or development systems although it will mention comparisons to some other common languages if the student has some prior knowledge of those. The student is assumed to be able to open PDF document pages and view and mark as favourites various important documentation sites available over the Internet, and an active Internet connection is assumed for at least the first few lessons, whereupon the student will be walked through the process of installing Elm locally such that an Internet connection will be necessary much less often (if a few help documentation pages are saved).

At the start, the lessons will cover the use of two different online Integrated Development Environment's (IDE's) that support Elm code, the first one as designed by the language's author and Benevelent-Dictator-For-Life (BDFL), Evan Czaplicki, and his team, which is linked from the main Elm web page with all kinds of other links to interesting things to read and including the link to the "Playground" IDE and the Elm Guide; the second is the Ellie-App built by one of the Elm aficionado's. Both only work with an active Internet connection whenever a new "compilation" (conversion of the source codes into a working web page) is triggered to the output as a separate frame in the IDE page. One might want to open all four of the above links in their web browser and keep them open during the beginning lessons, as they are a source of much of the information that will be taught, or contain links to such information. There will be another few web browser tabs worth of valuable links to documentation that will be useful as the student progresses.

Online Integrated Development Environment's

Now, with the above links open an a version of "Elm for Beginners" open on your computer, open the link for the simplest "Hello World" program in a new web browser tab (to open a link from a PDF file, one may need to hold the Shift or Control key while clicking it, and sometimes to open a link from a browser one needs to right click and "open in a new tab") to see the Ellie-app version of the "Hello World" program in all its glory! Before working through the explanation of the code, which is already in the "Beginners..." document, lets walk around the Ellie IDE to explain what it is we are seeing, as follows:

1. On the left hand side, is a row of icons that represent various things one can open with a click, of which there are only two that may be valuable to the student at this stage: The "Settings" icon with the gear opens up some options of which the only one that may be useful is the "Project Name" where one can assign a name for the project so that it will be part of the tab name as in "Ellie - Hello" if the project name is "Hello" as here and will be used if the project is saved or downloaded as a ".zip" file; and the "Package Manager" icon which looks like a package and lets one choose which packages will be used by the project, with the defaults of the "browser", "core", and "html", packages adequate for this simple project but to which has been added the "time" module that will be used at the end of the lesson. The "Package Manage" can be used to install and uninstall packages, but also to view the documentation for each of the individual packages on separate tabs if required. For instance, if one clicks on the "View Docs" link for the "html" package to open the documentation on a separate tab, then clicks on the "Html" link on the right hand side panel, one sees the "text" function documented on the left panel (some Type's, values, or functions may be further down the panel and require

scrolling or web browser find-in-page to locate) that explains the "text" function used in the "Hello World" program. One of the most important things about this documentation as well as telling us what it does is the "Type Annotation" to the right of it as in `text : String -> Html msg` that will be explained more fully later, but basically says that the function takes a text string and returns an HTML tag containing that string. The icons on the left toggle active/inactive on every click.

2. Along the top are some other necessary commands, again of which only a few are necessary at this stage: The "COMPILE" command that converts the source program from the left panel(s) into the output web page viewed when the "Output" button toward the far right is selected; the "RELOAD" command just rebuilds the web page without recompiling the code (often much faster) as a way of starting any recomputations from scratch. The Ellie-App has the means to generate a short link in the address bar for the current source code state, but while useful for documenting things will clutter up the Ellie-App server with saved source code indexed by links if used for every little change, as most browsers will remember the current state of the Ellie-App in the event of a power failure. The "Format Elm Code" code-looking icon on the right top corner of the left source panel converts the source code into "Elm standard" form (cosmetic only - doesn't change the meaning), which tends to put a bunch of extra space and lines into the code to make it more readable, which is fine but always "tab's" to four character stops which may tend to run the code far to the right for certain nested coding forms. The student can try it and if the results aren't to one's liking, can be reversed with the "Control-Z" key (pressed at the same time) to undo the change. This command is common the many editors, as are "Control-C" for copy selected text, "Control-X" for copy and cut selected text, and "Control-V" for past/insert copied/cut text at the current cursor (the blinking vertical bar). Also, the "Tab" key doesn't put tab characters into the source code but just inserts the appropriate number of space characters to the next tab position (every four characters).

3. The bottom of the two panels to the left is the HTML panel which contains the (mostly template) standard HTML file form to make the Elm code appear as if it was a standard web page, and which can be folded down the the down arrow to the right top corner. A coding beginner will not need to use this panel so can generally keep it folded down, but it can be used to show that HTML is nothing but some text is a specified format. For instance, if one wanted to show "Hello World!" in a web page in the same way as the Elm code is doing here, we could change the portion in the body to as follows:

```
<body>
  Hello World!
  <!-- <main></main>
  <script>
    var app = Elm.Main.init({ node: document.querySelector('main') })
    // you can use ports and stuff here
  </script> -->
</body>
```

1. to produce exactly the same thing without any coding! This "comments out" the link to the coding parts with the `<!--` and `-->` tags at the beginning and end of the tags to be ignored and adds the text in the body as the only element. All the part that makes the Elm code work is provide a "main" tag that the Elm code "hooks into" in order to generate its own web page, and has a couple of options in providing some JavaScript things to link into the Elm code and some "styling" things that aren't Elm code to style the web page in a standard way without coding. As a beginner, one won't be doing any

of those things, and the purpose of the above exercise is just to show that all the Elm code is doing after being converted to JavaScript is just generating HTML tags in a dynamic way (meaning that the HTML tags are changed as required by the web page) inside the "main" tag.

2. The top panel on the left side is the Elm "source" panel where we will be doing all of our work in editing our Elm source code and there is a left arrow just to the right of the "Format Elm Code" icon to fold both left panels to the left to give an almost full screen view of the right panel, usually containg the output web page.

3. There is a further feature in this IDE in that many package names and Type's, values, and functions defined in those packages will automatically open a link to the documentation for that item in the bottom left corner of the source panel above (just above the HTML in the bottom left panel), which if clicked will open a new tab with the documentation for that item in view.

The other "Try Elm"/"Playground" IDE works very similar to the above "Ellie-App" one with less capability in some respects and more in others. The major diffences are as follows:

1. There are only two upper panels, with the left one the Elm source code panel and the right one the output web page.
2. All of the links that can be clicked are along the bottom.
3. The leftmost Elm symbol just opens the main Elm web page as at the top of this lesson, from which other links can be selected.
4. The "Lights" icon just toggles between dark and light screens for the source code panel.
5. The "Packages" icon does the same as the "Package Manager" in Ellie but the `+` icons install a given package, the garbage pail icon removes the given package, and clicking on the name of a package opens its documentation page.
6. On the bottom right of the source panel is an icon a button that alternates between saysing "Rebuild", "Problems found" with error message(s) in the output panel, or "Success" and the resulting web page in the right panel; this is a button that can be clicked at any time to attempt another compile for the current source code.
7. The automatic help links are better than those of Ellie in that some/many Elm keywords also provide help links to the Elm documentation although this isn't complete.

With both of these online IDE's, the most reliable way of saving one's work permanently if the browser tab closes is to copy-and-paste the source code panel contents into a local text file which than then be copy-and-pasted back into a fresh invocation of the IDE. Also, the two IDE's are compatible so code can be copy-and-pasted between them as long as it contain the (normally optional) `module Main exposing (main)` line at the top required by Ellie-App.

Having now explained the beginners tools we will use, it is time to move on actual coding!

The "Hello World" Program and Variations

As well as the explanations in the "Beginner..." document, there are a few exercises we could do to play with this little program as follows:

1. Change the "import" line into the following (eliminating the `exposing` part) and try to determine from the error message what to do to fix it:

```
import Html
```

Answer: Now the `text` function by itself is no longer exposed so it can only be accessed via the "qualifier" of "Html" as in changing the last line as follows:

```
main = Html.text "Hello World!"
```

2. Change the "import" line into the following and try to fix it:

```
import Html as H
```

Answer: One should be able to solve this one from the error message.

3. Change the last two lines to the following and explain why it works:

```
import Html as H exposing (text)
main = text "Hello World!"
```

Answer: Although the "qualifier" for the package "Html" is now "H", no qualifier is necessary because the "text" function is also directly exposed

In Elm, we have "pipe" operators as in forward `|>` and backward `<|`. Explain what is happening with the following version of the last line:

```
main = "Hello World!" |> text
```

Answer: This is "piping" the String "Hello World!" to become the argument to the called function "text" to result in the same output. This could also be written as `main = text <| "Hello World!"` but the `<|` is redundant in this case.

The "Squares" Program and Variations

The "Squares" program is pretty well explained and variations suggested in the "Beginners..." document, but there is at least one simple variation that shows an algorithm can be built up in steps even though that may not be the most efficient, and then some (in this case small) optimizations applied later. This programming problem can be explained in words as "For the list of whole numbers from one to some bound, show a String of comma-separated bracketed pairs of numbers where the left number of each pair is the source value and the right is the squared value". We can break that down into steps in Elm code as follows:

```

```

```
main : Html Never -- Type Annotation that says what the `main` value
produces...
main = -- always required
  List.range 1 10 -- produces a list of Int's, which are whole numbers
    |> List.map -- to convert the simple list into a list of something else
         (\ n -> -- using this anonymous (no name) function that takes each
value
              (n, n * n) -- and turns it into a pair Tuple of the value and
its square
         ) -- end of function bracket
    |> List.map -- to turn the pair into a String
         (\ (n, nsqr) -> -- function grabs the first value in the pair as
`n` and the second as `nsqr`
              "( " -- new String starts with bracket space
                ++ String.fromInt n -- appends the String of the value of
`n`
                ++ ", " -- appends the comma space between them
                ++ String.fromInt nsqr -- appends the String of the value
'nsqr`
                ++ " )" -- closes the string with closing space bracket
         ) -- end of function
    |> String.join ", " -- function that takes a List of String's and makes
a string with given separator
    |> text -- resulting single String passed to make it into HTML.
```

Once this works, the first step of optimization was to combine the "map's" into a single map and produce the output List of String's in one step; other optimizations may be possible...

Now, the next program is a radical departure from this example: In both variations of the above program, Higher Order Function's (HOF's) have been used with build in functions, which is often the easiest to visualize and understand but may not be the fastest of most efficient. The following code uses direct recursion to implement the program by the use of a local function to do the "loop" without using mutable variables:

```
import Html exposing (Html, text) -- import HTML for the Type Annotation
main : Html Never -- Type Annotation that says what the `main` value
produces...
main = -- always required
  let -- a local "block" up to the `in`...
    loop n rstr = -- a local function to be used recursively (calls itself)
through tail call...
      if n > 10 then rstr ++ " )" -- if done, return the result string with
closing space bracket
      else
        if n <= 1 then loop (n + 1) (rstr ++ "( " ++ String.fromInt n ++ ",
" -- first loop, no prefix
                                          ++ String.fromInt (n * n) ++ "
)")
        else loop (n + 1) (rstr ++ ", " ++ "( " ++ String.fromInt n ++ ", "
```

```
  -- rest loops, comma space prefix
                                          ++ String.fromInt (n * n) ++ " )")
    in loop 1 "( " |> text -- call the local function and pipe the result to
  "text"
```

This little program demonstrates how we overcome not having any mutable variables in functional languages like Elm: we create a function that calls itself/recurses with new values for the arguments for every call, which to the program logic looks like the same value changing but actually is a new invokation. Internal to the generated code being run, this is often converted into mutation, but we are safe from having to handle such an ugly thing - the dirtiness is handled for us in a safe manner. So `loop` starts with the value of `n` of one and the result string as just the opening bracket space, then for every loop where the value of `n` is less than the limit of ten, it creates a new value of the result string (with the first loop not prepending the separator and subsequent loops appending it) as well as calling itself with the new incremented value of `n`. These (two) calls to itself (the `loop` function) are in tail call position because no further use is made of the values returned by the function after the function call returns (ie. the values returned aren't passed to another operator or function). This code is about as short and likely faster than the previous version because there is no List manipulation going on. In fact, the `List.range`, `List.map`, and `String.join` function will be using tail call recursions internally in exactly this form but separated, so there will be loops followed by loops, etc., which is yet another reason it is slower. However, for many applications the clarity of using HOF's is worth the loss in efficiency, and for many applications (as here), the speed doesn't matter.

The "Bouncing Ball" Program and Variations

This "Bouncing Ball" program as well as some ways to do variations is quite well explained in the documentation, but there are some details that are not explained, as follows:

1. In this program, we have exposed everything as in `(..)` from the `Playground` module, but as explained further down the "Beginners..." document, we could have exposed only what is actually used or qualified everything, optionally with a shorted alias for `Playground` say as `PG`.
2. This program does not use Type Annotations for the global values `ballRadius` and `main` nor the global function `view` as is the recommended style for globals and is covered further down the "Beginners..." document.
3. If we look up the necessary Type for `ballRadius`, we see that it must be a `Number` because the functions to which it is passed as an argument expect a `Number` but we see that `Number` is just a `type alias` for the built-in `Float` Type which is a 64-bit JavaScript float value.
4. The call to the `game` function for `main` must pass it two functions, one that handles the inputs and the other that handles the conversion to a view; since we have no user inputs, there isn't any updating of the computer state due to these inputs and the update function is then just a simple anonymous (no-name) function the ignores its inputs (the `_` characters) and returns the "memory"; however, as "memory" is never used it can be anything so instead of a Tuple of `(0, 0)` used here, could be "nothing" as in `()` or an empty Record as `{}` or an empty List as in `[]`. Try these and you'll see it still works the same. The "nothing" value of `()` is preferred in these situations because that is literally what it means.
5. The local bindings of `left`, `top`, `right`, and `bottom` are local (`let...in`) values that only exist within the `view` function and have the values as computed from the given current `computer.screen` structure and the given `ballRadius` constant value in order to subtract the ball

radius from the actual current screen dimensions. These calculations could have been embedded in the calls to `zigzag` as they are each used only once, but it makes the code much clearer this way.

6. The `view` function must produce a `List` of `Shape`'s which `Shape` type is defined in the `Playground` package, and in this case is the returned value of the `circle` and the `move` functions. If you click on these functions and then click on the resulting links at the bottom of the source code panel, you can read the documentation for these as to what Type's they take as input and what they return. The enclosing `List` is the result of the calculation inside the square brackets `[...]` which is the `List` literal as explained further down in the "Beginning..." document.
7. In this program, the `circle` function is producing a `Shape` centered on the screen which is then "piped" to the `move` function to move the `Shape` into a new position according to what two applications of the `zigzag` function produces for `x` and `y` coordinate `Number`'s by producing a new `Shape` with those new coordinates (remember, no mutation).

As one can see, there is quite a bit to be learned from even this simple animation program.

In Lesson Two, we will explore a preview of building simple HTML pages with keyboard or mouse input using Elm using the examples from the IDE's...