Elm for Beginners

Elm is a programming language used to build web pages. This document is a guide for potential users of the Elm programming language who do not necessarily know any programming languages to be able to write basic programs in Elm to build their own web pages that can be sent to and used by anyone through something as simple as just sending the generated HTML files to others or serving them as web pages from their Google Drive account to submitting them to (many free) web page servers.

Simple examples of Elm Code

The very simplest program in any language is one that just prints "Hello World!" to the screen; the Elm version is just two lines, as follows:

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

which makes a web page that contains nothing but the "Hello World!" text.

This link to an online Integrated Development Environment (IDE) shows this in action from the program in the left hand panel generating the web page shown in the right hand panel after "compilation", which is just yet another computer program translating the text of the program to the (actually just different text) HTML web page. The additional line at the top `module Main exposing (main)` is an additional requirement of

this particular IDE and not strictly required by the Elm language itself, although recommended. Now this isn't very interesting as one could generate the required minimum HTML code to do the above as follows:

```
<div>Hello World!</div>
```

which if saved using a text editor to a text file named something like "index.html" will do exactly the same thing when opened with a web browser!

So let's do something a little bit more interesting - perhaps a little mathematics: How about calculating all the squares of all the numbers from one to ten? The following Elm program will display the results of that calculation (with the "module" line included this time as required by the IDE):

```
module Main exposing (main)

import Html exposing (text)


main =
    List.range 1 10
        |> List.map
            (\n ->
                "( "
                    ++ String.fromInt n
                    ++ ", "
                    ++ String.fromInt (n ^ 2)
                    ++ " )"
            )
        |> String.join ", "
        |> text
```

with this time the program formatted to not be just "one-liners" so as to make it easier to read and understand as per this link to an IDE version. It likely isn't too hard to guess what number to change to make this program output the squares of the numbers from one to a hundred or one to a thousand or what numbers to change to output the range from 9990 to 10000. The current program just outputs the following, of course:

```
( 1, 1 ), ( 2, 4 ), ( 3, 9 ), ( 4, 16 ), ( 5, 25 ), ( 6, 36 ), ( 7, 49 ), (
8, 64 ), ( 9, 81 ), ( 10, 100 )
```

with the input number to be squared on the left and the square on the right of each bracketed set in the series.

The above program wouldn't be quite as easy to just generate the HTML code for (including the calculations) so one starts to see the point of using a programming language. The code is fairly easy to read and understand as a "flow" (using the "forward pipe" operator symbol `|>`) from making a list of the numbers in

the range between the limits, to mapping that to a new list that contains a text "string" of the results (the calculation of the square and the generation of the result "string" containing the original number and its square is combined to keep the code short, but this could have been two separate steps), the elements of this "string" list are combined/joined into one text "string" separated by a comma and a space, and finally the result string is sent to form an HTML text "tag" just as for "Hello World!". The thing to remember in reading code using the "forward pipe" operator `|>` or the "backward pipe" operator `<|` is that they take the source "thing" on the left and right, respectively, and pass it as the last argument of the destination function on the other side; thus, here it is taking the `List number` generated by the `List.range` function and passing it as the thing to be mapped by the `List.map` function and its mapping function (anonymus function used), then passed as the `List String` to be used by the `String.join` function that puts `", "` between each element and returns just a `String`, to finally pass it to the `text` function that generates the HTML code from the `String`.

Now the above program just generates a "static" web page that doesn't change once the calculation is done (once) and the result is displayed. How about something more interesting that changes? The following program shows a bouncing ball randomly bouncing around the page:

```
module Main exposing (main)

import Playground exposing (..)


ballRadius =
    40


main =
    game view (\_ _ -> ()) ()


view computer _ =
    let
        left =
            computer.screen.left + ballRadius

        top =
            computer.screen.top - ballRadius

        right =
            computer.screen.right - ballRadius

        bottom =
            computer.screen.bottom + ballRadius
    in
    [ circle green ballRadius
        |> move (zigzag left right 4 computer.time)
            (zigzag top bottom 3.6 computer.time)
    ]
```

as shown in this link. As to how it works, this time an extra package has been imported that does the graphics as in "evancz/elm-playground" from which we run the `game` mode (in order to obtain the screen size parameters, with the `memory` parameter not used and starts as `()` and if changed would change to `()` - no change); in the `view` function of the `game` mode a green circle/ball of a given radius is drawn (at the center of the "screen") and moved by the `move` function to a position as given by the horizontal and vertical "x"/"y" co-ordinates passed to it as its first arguments, which arguments are as produced by the time based `zigzag` function calls with the limits of the screen reduced by the radius of the "ball" to keep the ball within the "screen" limits; two different "zig zag" rates are given so the "ball" follows what appears to be a random bounce pattern. It should be easy to see what to change to change the radius and color of the "ball" and the bounce rate in each of the `x` and `y` directions. Due to reading the computer screen sizes every time the view is updated, the program is still correct even when the screen size is changed while the program is running.

As one can see in the above examples, especially the last one(s), one can accomplish quite complex programming tasks with very little Elm code, with the code easy to read and understand. However, any difficulty is in the details and knowledge required to write and test that code as in the following sections.

Notes about "Significant White Space"

Like the functional programming languages of Haskell and F# and the imperative language, Python, Elm is a "significant white space" language where indentation is significant, but at least for Elm, indentation isn't very "significant" and comes into play in only four places, as follows:

1. Every definition within the global structure must start on column one (the first column) and everything up to the next definition must be indented at least one more column. This includes the module definition, each of the imports, and the type definitions, type annotations/signatures, and the value/function definitions as will be defined in the next section on structure.
2. For `case` statement targets, the branch target patterns for a given `case ... of` need to all be aligned **but they do not need to be aligned with anything in particular** other than being indented at least one column from the outer block "scope"; then the `->` symbol and the expression for each target pattern needs to be indented at least one more column from the pattern for which it applies
3. For local definitions inside `let ... in` blocks, the same rules as for rule 1 apply except the alignment indentation is set to whatever column that the name of the first value/function definition starts which is to the right of the `l` in `let` instead of column one.
4. No white space nor block comment may contain "tab" ('\t') characters but they are accepted in line comments as they can't affect indentation there. Most text editors used for programming have a setting so that pressing the Tab key doesn't output a Tab character but rather the number of spaces so that the indentation aligns with the next indentation level as per the settings, usually either every two or four spaces; thus, with this setting Tab characters will never be inserted into the "source" text.

Elm comments are treated as white space for the purposes of calculating indentation point and are defined as follows:

1. Line comments start at the `--` symbol and continue to the end of the current line.
2. Block comments start at the first `{-` symbol and end at a matching `-}` symbol but may be nested meaning that there must be as many close symbols as open ones. Proficient Elm programmers use a little "hack" to quickly "comment out" blocks of code by putting a `{--}` just before the block and a `--}` just after the block, neither of which do anything except change indentation as the first one is both opening and closing block comment symbols in one and the second looks just as a line comment with

the leading double-dash; then, when one wants to comment out the block, they can just insert a space in the first one to make it `{-- }` which makes the closing block comment symbol ineffective with the double-dash then ignored because it is now inside the block comment and the final closing block comment symbol coming into play. Due to Elm's block comments being nestable, this works even when there are block comments within the block being "commented out".

The following (ugly) Elm code demonstrates how relaxed Elm's indentation actually is as the following is valid Elm code:

```
{-
Ugly
Elm
-}
module
 Main
 exposing
 (
 main
 )
import
 Html
 exposing
 (
 text
 )
test
 n
 =
 case
 n
 of
 0
  ->
  True
 _
  ->
  False
main
 =
 let
  r
   =
   test
   42
  s
   =
   Debug.toString
   r
 in -- the `in` can be on any column from this one right...
 text
 s
```

where the main points are that a step-in of indentation is only required for the block inside a new definition, for the sub definitions inside a `let ... in` block (with all name definitions aligned to the first name), for everything after the name of a definition, and for everything after the pattern expression of the target of a `case ... of` (with all target patterns aligned, but the alignment can be actually stepped out from the `case` although to no more than one column inside the outer definition), and that the expression after the `in` can be stepped out all the way to one more than the indentation of the enclosing block.

Now one shouldn't write Elm code that way and should use some sort of styling guide but it shows that "significant white space" isn't very "significant" in Elm.

The Characteristics of the Elm Language

Elm is a statically-typed purely-functional programming language; as a beginning programmer, those two characteristics may not mean much but generally have the following significance:

1. The first characteristic of being "statically-typed" clearly means it can't be "dynamically-typed" as many languages such as Python are and means that all the Type's of all values and function parameters and result must be known when the program is compiled (remember: converted from the source code form to its runnable form) rather than being determined at run time as for "dynamically-typed" languages. Static-typing has been shown to produce much more reliable and easily tested code than dynamically typed code which more than makes up for the extra effort required to get the types right as the code is written. For the most part (as used in the above examples), Elm's type inference during compilation determines what the static types must be from the way values and functions are used in the code, but there is provision for providing clues to both the compiler program and the user as to what the types must be in the form of Type annotations/signatures as explained later.

2. The meaning of being a "purely functional" language is a very complex subject but generally means that, being "functional" that all elements in the code are either values of Type's or functions that take at least one value (which may be a function) and return a value (which also may be a function); being "pure" generally means that all values are "immutable"/unchangeable upon first creation at least as far as the programmer is concerned: what this means as to "safety" of Elm programming is that there are limited well-defined ways to manipulate the values and functions used by a program so that the program is "safe" in not enabling one part of a program to change a value while another part of the program depends on its old value. In short, "pure" functional languages such as Elm only have two types of programming entities: immutable values which don't have parameters/arguments, and immutable functions that have one or more defined parameters/arguments (parameters are as per the function definition; arguments as as when the function is applied/called).

The Structure of an Elm Module

As one can see in the above examples, Elm code follows a very simple (compulsory) format, as follows:

1. A program starts with the (normally optional for applications such as these) `module` definition which gives the capitalized name of the `module` (which must match the name of the file of this source code including case and excluding the ".elm" extension) and the list of values and/or functions and/or Types that are defined and exposed by the module (at least a `main` value - not function - when the module is the start of an application) For Types, the programmer can choose to expose only the Type or in the case of Custom Types as explored below, can choose to export the Type along with it's

Constructors as in CustomType(..). This definition must be the first non white space (spaces and new lines are "white space")/non-comment code in the source file and must start on the first column.

2. Next, a module must list any imported modules from packages that are listed to be used in the project "JSON" file named "elm.json" as a project definition file, with an optional `exposing` list of the value's, function's, and Type's used by the module where if they aren't "exposed" they must be used as "qualified" by the imported module name as in `Html.text`, `Playground.game`, etc.; symbolic operators must be exposed to be used as qualification doesn't work with them (no `List.(::)` or `List.::` can be used). Also, in order to be imported and exposed, Type's (and their constructors), values, and functions, must be exposed by their respective defining module in the respective `module` line as above. As an option, if the module Name (which may have a lot of dot sub modules in it) is too long to be used conveniently, a "shortcut" alias name may be specified with use of the `as` keyword after the Name and before the `exposing`, if used. Again, these must be separated from the `module` definition only by white space and comments and each `import` must start on the first column. For instance, the bouncing ball program could uses and of the following forms of imports that would do the same thing:

```
import Playground as PG -- requires everything that uses functions from the
module to use `PG.` such as
                        --   PG.game, PG.computer, PG.circle, PG.green,
PG.move, PG.zigzag, etc.
import Playground as PG exposing (computer, zigzag) -- now `computer` and
`zigzag` can be used without the `PG.`
                                                    --   rest as above...
import Playground exposing (..) -- as per the example, everything exposed
by the module is known without qualification
```

3. Finally, in any order are the main module Type, value, and function definitions (where value and function definitions may have type signatures/annotations preceding them at the same indentation level); in the above examples there is only a `main` value definition and a `ballRadius` value in the last one and no new Type's or functions are defined but they generally follow the same form of having any amount of white space and comments between them but must start on the first column. An example of a value type signature and a function signature for the bouncing ball example are as follows:

```
ballRadius : Number -- `Number` is a type alias for `Float`, which in Elm
is a 64-bit float...
ballRadius = ... -- exactly as before, as the type signature just verifies
what the compiler already knows...


-- `Computer` is a type structure defined in the `Playground` module to
provide screen size and time, etc.
-- `memory` represents anything at all as it isn't used in the function as
indicated by the `_`...
-- the return value from the function is a list of `Shape`'s, which
specification is again
```

```
--   defined and exposed/exported by the `Playground` module...
view : Computer -> memory -> List Shape -- this is the annotation for a
function as per the `->`'s
view computer _ = ... -- exactly as before, as the compiler will have
already determined the above Type's...
```

Elm Types

Elm types can be generally grouped into two sub classes as in simple Type's and complex Type's, where the distinction between the two quite is easy to make: simple Type's are those describing things whose storage can fit into one computer register where complex types cannot. The size of computer registers is somewhat variable depending on the CPU, but for all modern systems upon which Elm code is likely to be run, there are either 32-bit CPU's with 32 bits at eight-bits-per-byte or four bytes per register or 64-bit CPU's with eight bytes per register; all current CPU's also have built-in floating point processors with their own floating point (that can have decimal point fractions and optional exponents) registers of at least 64-bits. In Elm, all Type names must start with a capital letter and (if we were defining our own types) followed by any number of numeric or upper or lower case letters interspersed with _/underscore characters.

Simple Elm Types

From the above qualification of taking no more than one register's worth of space, one can see that the following types qualify as simple types:

1. The `Float` Type is actually the only "native" JavaScript numeric type and generally will be used by putting it into one of the CPU's 64-bit floating point processor registers.
2. The `Int` type is not a real JavaScript type but is generally just using a sub range of the `Float` type to represent 32-bit integers that can be represented within the 64-bit `Float` mantissa.
3. The boolean `Bool` Type is included here in spite of that Elm may implement it so it takes a little more than one CPU register to store it; however, that is just an implementation detail as it could easily have been implemented to take the same space as an `Int`. The `Bool` type has two values: `True` or `False` and must be either one or the other of those.
4. The `Char` type fits into an `Int` type as it describes a UTF-8 variable-length structure between one and four bytes long (depending on value) with encoded values from zero to 1,114,111 (just over 20 bits worth).
5. The "nothing" type which doesn't have a name in Elm is represented as `()` like a tuple without any contents and has a single value of `()`. This Type/value doesn't actually need to use any storage at all but in the current Elm implementation actually does take some small amount of storage. In some languages this may have names such as the `None` type in Python or the `Unit` type in some functional languages.
6. The Never type is a type that can never be used, which sounds to be quite useless but in Elm completes the Type system in representing types that are never used. For instance, for the first two examples above, the `main` value has the type of `Html.Html msg` where the `msg` type variable is never referenced; thus this could just as easily be identified as having the type `Html.Html Never`. The `Never` Type can be paired with the `never` function that can never be called as in it takes a `Never` type and would return any other type if it were actually called but the `never` function is used just to make the Type system happy in allowing the conversion of a type that contains a `Never` to a type that has something else in the place of the `Never`; for instance if a module that produced an `Html.Html`

`Never` Type were imported and used by a module that actually used the `msg` type, the never conversion function could be used to indicate that the two types are compatible as explained in the core package `Basics` documentation. Like the "nothing" type, this type doesn't need to use storage but in current Elm it does use some storage.

Complex Elm Types

The complex types are as follows:

1. The `String` Type is some kind of a collection of any number of UTF-8 `Char`'s processed as a fixed/"immutable" group so obviously can't generally fit into a CPU register.
2. A tuple Type is a comma-separated group of two or three possibly different Type's inside round brackets as in `(42, "hello")` or `(123.4, 'a', (42, "hello"))`, which shows that the types can be all different and that a tuple Type can nest any other Type's including another tuple Type. Each combination of contained types is a new tuple Type instance containing exactly those given types and distinct from other tuple Type's containing other types. The `()` "nothing" type is not a tuple Type as it is a type unto its own nor is a single type enclosed in round brackets a tuple Type with one "field" as the round brackets without enclosing a comma only forms a (possibly redundant) grouping of the inner single Type.
3. A record Type is a comma separated group of any number (zero on up) of **named** fields of possibly different Type's inside curly brackets as for example `{ num = 42, str = "whatever" }` and as for tuples can contain any other type including other records of other types or the same type. Also, as for tuples, each combination of contained types is a new record Type instance containing exactly those given types with those named fields and distinct from other record Type's containing other types and/or of different field names.
4. A List type is a "linked" list of any number from zero up of elements all of the same Type and again, a List containing particular types of elements is distinct from all other List's containing other Type's; as to what "linked" means, each element node of a list contains a "head" value of the given element type and a "link" which is another List of the same element Type, but with List's, a given node may be empty. List literals are then just a comma separated list of the same type of values inside square brackets as in `["please", "be", "good"]`, for example, or the empty List as with only one value for all Type's of List as in `[]`
5. The super type that could build all other Type's is the Custom Type, although some of the other types have some computer built-in "magic" abilities in order to manipulate them more easily. A Custom Type is a general container Type which can include any number of different types with the characteristics of record fields (although unnamed, which naming is part of the compiler "magic" for records) and are therefore like tuples when only having two or three unnamed "fields", but are much more than that as they can have any number of "Constructors" (capitalized functions) that build variants that each can contain any number of (unnamed) data "fields" of any Type. Thus, the `List a` Type where `a` represents any given type can be defined using a Custom Type as `type List a = Cons a (List a) | Empty` where `|` means "or" and with "computer magic" linking up `[]` to mean `Empty` and construction "magic" to cause the linked chain of element nodes to be created from the literal forms such as given above, as well linking the "cons" symbolic operator of `::` to the `Cons` constructor. An even easier commonly used type that is made using a Custom Type is `Bool` which is defined as `type Bool = False | True` or in other words, a Custom Type with two Constructors named the two `Bool` values of `False` and `True`. The Custom Type's can contain any other Simple or Complex type including itself as used in the `List a` definition above.

6. The Function Type is included in the Type system as a valid type that can be used as fields of another type, as a value, or passed as arguments to any function, where in Elm as in all functional languages, a function is something that takes one or more functions or values as arguments and returns something which may be a value or function. Elm functions can be named or anonymous (unnamed) but if created as anonymous functions are usually bound to a name as a value definition or as a named function parameter so become as named functions as far as use goes. Functions may have Type annotations that express what types they take as parameters to be possibly passed to their result expression that produces the function result Type but whether this annotation is present or not, there is an internal type signature that the compiler has inferred. All functional language functions must return a result (and returning `()` would be useless as it would indicate that the function could only be called for its effect where "pure" functional languages don't have effects). For example, the following definitions:

```
outerFunc() =
  let o = 57
      testFunc1 x y = o + x + y
      testFunc2 = \ x y -> o + x + y
  in testFunc1 7 42 -- testFunc2 7 42 produces exactly the same
```

produces exactly the same acting inner functions although they are separate. In addition, the two functions are "closure's", which means that each captures something from outside that is not a parameter to the function as in the `o` value in this case.

7. There is a special "shortcut" creator as in `type alias` which can create a (capitalized) shortcut Name for any Type, but with the limitation that these "shortcut" Names cannot be used recursively; in other words, one could not create a List type from tuples or records just by referring to themselves using one of these "shortcut" names as in `type alias List a = (a, List a)` because the recursion causes the Type to be unsolvable, whereas it is solvable when a Custom Type is used because the Constructor acts as a function with the recursion inside it which breaks the "loop". Only for records, one of these shortcut Names pointing at a record type can be used as for a Custom Type Constructor where no curly brackets and field names must be used to construct the record Type but only the actual values are used as arguments in order, as in the following:

```
type alias MyRecord = { num : Int, name : String }
val1 = { num = 42, name = "John" }
val2 = MyRecord 42 "John" -- exactly equivalent results!
```

Type "variables"

In Elm, Type aliases and Custom Types have a way of representing what other languages call "generic" Type's: defining Type's to work with any associated Type(s). This is used in the `List a` definition, for example, in that the definition says that `List`'s can contain elements of the constant type represented by `a` where the actual type is assigned to `a` when the `List` type is actually used in code. The `Result x a` Type uses two Type "variables", with `x` representing the Type returned with something fails and `a` representing

the Type returned when it succeeds. Custom Type's can be defined with any number of Type "variables" or none.

Constraints on Type "variables"

Elm does not have a way of defining "Type Classes" as in Haskell that can constrain/limit what Type's a Type "variable" can represent and what functions can use those Type's as arguments dependent on whether it is defined as an instance of that "class"/interface or not, but it does have a limited way of designating that a Type "variable" represents a constrained/limited set of `Type`'s, as follows (each of these may have a number appended to it to distinguish it from ones with a different number so that `number0` might represent an `Int` everywhere it is used in a Type annotation/signature and `number1` might represent a `Float` in a particular use where if only `number` were used, all uses would have to be the same Type):

1. The `number` constraint says that the actual Type used in place of `number` must be an `Int` or a `Float` and there are a whole set of functions defined in the `Basics` module that is always imported automatically that applies to `number`'s.
2. The `comparable` constraint says that the actual Type used in place of `comparable` must be a number, `Char`, `String`, tuple, or a `List`, which defines what types work with the `(<)`, (<>=), (<=), and `(>=)`, operators. Notice that record Type's and general Custom Type's are not included here so cannot be directly compared although their fields may be.
3. The `appendable` constraint says that the actual Type used in place of `appendable` must be a `String` or a `List` which defines what types work with the `(++)` operator.
4. The `compappend` constraint says that the actual Type used in place of `compappend` must be a `String` or a `List a` where `a` is the set of `comparable` Type's from item 2 above and to which both the `comparable` operators and the `(++)` operator can be applied; this constraint is rarely if ever used in real programmer code but may be used internally by the Elm compiler.

Type Annotations or Type Signatures

An optional declaration but highly recommended for global values and functions is a Type Annotation or Signature, which has the same indentation level as the name of the value/function and is immediately above the value or function definition separated from it only by white space and/or comments. These use the special symbol `:` to show that the preceding value/function name plus the parameters that all functions have at least one of have the Type defined following the `:`, where the only other use of the `:` is to assign Type's to record fields as Type signatures in exactly the same fashion as seen in the example record code above. When the definition is for a function (with parameters representing arguments), the `->` is used to separate each parameter one from the other and from the final result Type of the function.

So, for example, in the first and second examples above, the imported `text` function has the following type signature: `text : String -> Html msg` meaning that it takes a `String` as a single argument and returns an Html element, with the `msg` representing some sort of arbitrary message Type which may be required by other code. In these two examples, this `text` function produces the result for the `main` value so `main` must have a type signature of `main : Html.Html msg`; however, since these two example programs don't actually use a message, then the `msg` can be replaced by anything, including "nothing"/`()` or `Never` as it is never used.

For the third graphics example, the `ballRadius` has to be a `Number` which is defined in the `Playground` package as a `type alias` for `Float` in order to be a proper argument for `circle` so its type signature

should be `ballRadius : Number` or `ballRadius : Float` and the `main` value needs to have the same type as the result of `game` from the `Playground` which as used here means `main : Program () (Game ()) Msg` except that neither `Game` nor `Msg` are exported by the `Playground` module so we can't use a type signature here and need to just let the computer figure it out automatically (this is a fault with the definition of the "Playground" module). The type signature for the `view` function as required by the `game` definition is `view : Computer -> memory -> List Shape` but since we use `memory` as nothing/`()` and given that both `Computer` and `Shape` are exported/imported from the `Playground` module, we can define this type signatures as `view : Computer -> () -> List Shape`. The `circle Shape` is passed to `move` which returns the same `Shape` just with a new location, and that is wrapped inside a single element List of `Shape`'s so it satisfies the above type signature/annotation.

Type signatures are not usually supplied for internal value and function definitions as here for `left`/`top`/`right`/`bottom`, but a common use of internal annotations/signatures is for numbers which may be either `Float` or `Int` when it is desired to force a value to one or the other; in the Bouncing Ball example case, they must be `Float` because they are used by functions that expect `Number` aliased to `Float` so a type signature are not necessary. These internal type annotations/signatures must be aligned with the definition name to which they apply and be separated from them only by white space and/or comments just as for use on global values and functions.

Advanced Use of Custom Type's to Define Any Other Type

As demonstrated above, one can generally define about any Haskell type using combinations of the above general types where if one wants to have named fields, one can just wrap a record in a Custom Type although that isn't really much different than just a `type alias` for the record. Now Haskell has possibly infinite lazy lists, so as a first pass at defining those we might define an infinite Co-Inductive Stream (CIS) as follows, where the difference between CIS's and true lazy lists is that there is no memoization for CIS's so that the calculaton is guaranteed to be run only once, whereas true lazy values only have their defining function "thunk's" run once with the value saved internally to be used for all further references:

```
module Main exposing (main)

import Html exposing (text)


type CIS a
    = CIS a (() -> CIS a)


headCIS : CIS a -> a
headCIS (CIS head _) =
    head


tailCIS : CIS a -> CIS a
tailCIS (CIS _ tailfunc) =
    tailfunc ()


takeCIS : Int -> CIS a -> List a
```

```
takeCIS n cis =
    let
        loop i (CIS head tailfunc) revlst =
            if i < 1 then
                List.reverse revlst

            else
                loop (i - 1) (tailfunc ()) (head :: revlst)
    in
    loop n cis []


naturals : () -> CIS Int
naturals () =
    let
        next v =
            CIS v <| \() -> next (v + 1)
    in
    next 1


main : Html.Html Never
main =
    naturals () |> takeCIS 10 |> List.map String.fromInt |> String.join ",
" |> text
```

which produces the following:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

The above code works well enough because each new value of the sequence only needs to refer to the last and not further back than that. Let's change it to output the Fibonacci sequence which is the series of numbers which are the sum of the previous two with the frequent Haskell implementation just using the zip of the sequence itself with the same sequence with one element dropped from the first as expressed in Haskell as follows:

```
fibs :: [Integer] -- this is a list of "Big Integers" in Haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

This code can be explained as follows: given that `fibs` represents the to-be-produced lazy list of Fibonacci numbers (meaning it can be "infinite" as it is extended as required), the list is defined with the leading zero and one (so the later functions don't run away into infinite recursion because they are using the heads of the List that hasn't been defined yet), the `zipWith` function takes the "plus" function, the `fibs` list, and the `fibs` list with the first element dropped, to first produce pairs of the the lists in order and then the sum of the elements in those pairs, which is the Fibonacci sequence.

The above can be expressed in Elm as follows (Elm has a "bigint" module that could be imported - not used here):

```
module Main exposing (main)

import Html exposing (text)


type CIS a
    = CIS a (() -> CIS a)


headCIS : CIS a -> a
headCIS (CIS head _) =
    head


tailCIS : CIS a -> CIS a
tailCIS (CIS _ tailfunc) =
    tailfunc ()


takeCIS : Int -> CIS a -> List a
takeCIS n cis =
    let
        loop i (CIS head tailfunc) revlst =
            if i < 1 then
                List.reverse revlst

            else
                loop (i - 1) (tailfunc ()) (head :: revlst)
    in
    loop n cis []


zipWithCIS : (a -> b -> c) -> CIS a -> CIS b -> CIS c
zipWithCIS func cisa cisb =
    let
        next (CIS heada tailfunca) (CIS headb tailfuncb) =
            CIS (func heada headb) <| \() -> next (tailfunca ()) (tailfuncb
())
    in
    next cisa cisb


fibonaccis : () -> CIS Int
fibonaccis () =
    CIS 0 <|
        \() ->
            CIS 1 <|
                \() ->
                    let
```

```
                        firstfibs =
                            fibonaccis ()

                        nextfibs =
                            tailCIS firstfibs
                    in
                    zipWithCIS (+) firstfibs nextfibs


main : Html.Html Never
main =
    fibonaccis ()
        |> takeCIS 30
        |> List.map String.fromInt
        |> String.join ", "
        |> text
```

outputting the following:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
317811, 514229
```

but with a major problem due to the use of CIS's as to performance: because every new value of the sequence depends on every other value of the sequence back to the beginning, the performance is factorial with increasing length and builds huge execution stacks in memory due to the back recalculation where for larger problems it will run out of space, whereas in Haskell with every element only calculated once and "memoized", it takes linear time with length and very little space is used with the sequence consumed as it is used.

While it is true that there are better iterative algorithms for this sequence that don't require back calculation, it shows a limitation in not allowing a lazy memoization as Elm version 0.18 had at the minor cost of a `Lazy` module in the `Core` package and a warning in the documentation thereof that, while elegant, the use of lazy memoization is often slower than using iterative recursive algorithms.

For this particular case, the iteration by recursion approach fixes the performance problems as per the following code:

```
module Main exposing (main)

import Html exposing (text)


type CIS a
    = CIS a (() -> CIS a)


headCIS : CIS a -> a
```

```
headCIS (CIS head _) =
    head


tailCIS : CIS a -> CIS a
tailCIS (CIS _ tailfunc) =
    tailfunc ()


takeCIS : Int -> CIS a -> List a
takeCIS n cis =
    let
        loop i (CIS head tailfunc) revlst =
            if i < 1 then
                List.reverse revlst

            else
                loop (i - 1) (tailfunc ()) (head :: revlst)
    in
    loop n cis []


fibonaccis : () -> CIS Int
fibonaccis () =
    let
        next back2 back1 =
            let
                newvalue =
                    back2 + back1
            in
            CIS newvalue <| \() -> next back1 newvalue
    in
    CIS 0 <| \() -> CIS 1 <| \() -> next 0 1


main : Html.Html Never
main =
    fibonaccis ()
        |> takeCIS 30
        |> List.map String.fromInt
        |> String.join ", "
        |> text
```

with the same result indicating that the use of the `zipWith` algorithm is likely not ideal anyway, but it does show that not having memoization takes away some options that may be valuable for some algorithms.

As one can see, even if the CIS's and/or lazy list types and functions were defined in packages so they wouldn't have to be defined to be used as here, Elm coding style generally has more white space than does often used "compact" Haskell coding style, but this also makes the code more readable; other than the differing styles and the more descriptive words used for names of values and functions, the code would be very similar.

Literals for Each Elm Type

Literals are constants of the desired type, so for each Elm Type defined above, the following are the ways to define literals/constants of that Type that can be used in Expressions:

1. `Float` literals are any number written with a decimal point and/or an exponent such as `1e1` which is the same as `10.0`, but being forced into being `Float`'s due to the exponent and decimal point, respectively.
2. `Int` literals/constants are more difficult to define non-ambiguously because they will natively be Type'd as just numbers which could be either `Float` or `Int`. In order to force a given number without a decimal point to be used as an integer, one needs to define it with an overall Type annotation/signature as a separate entity or apply a functions such as `truncate`, `floor`, `ceiling`, or `round` to the number that will then force the number to be a Float converted to an Int by various individual conversion rules which all turn out to be the same when the number is an exact whole value in the first place. Another alternative is to define the number in hexadecimal as in `0x2A` which is the same as 42 in decimal but is always an `Int`. Also, some operations/functions only take arguments that are `Int` so if the number is used with such, they will be forced to be `Int` by the compiler type inference process.
3. `Bool` literals are formed as one of two values: `False` or `True` as per their definition as a Custom Type, which shows that, depending on implementation details, Custom Types can be used as primitive Type's - When they consist of only Constructors with no data for any one as here and could be used as things that have a sequence, or when they have only one Constructor which has only one data field which is a primitive type in which case they become a distinct Type wrapper for that data type and the wrapping need not take any extra memory space.
4. `Char` literals are inside a pair of right parenthesis as in `'A'`, but the single letter may be escaped to represent a limited set of "control" characters as in `\r`, `\n`, `\t`, `\"`, `\'`, and `\\`, which represent line return, new line, tab character, the double-quote character (used when it can't be written directly as inside single strings), the right parenthesis character (used when it can't be written directly as inside `Char` literals), and the back-slash character (which can't be used otherwise as it is used to form these "escapes), respectively. All printable characters can be input directly even if they aren't represented on a particular keyboard used, but any UTF-8 character can be used by "escaping" it as a unicode escape sequence, `\u{xxxxxx}`, where the `x`'s represent a hexadecimal number that can be any hexadecimal character (0-9/`a`-`f`/`A` - `F` where any of the letters can be upper or lower case) from four to six digits long, and where the value represented is between zero and `0x10FFFF` inclusive.
5. `()` literals are themselves or `()`.
6. Since the Never Type can never be used, it has no literal value.
7. `String` literals are a group of `Char` values inside either a pair of double-quote characters as in `"hello"` or a pair of triple double-quote characters as in `"""hello"""` where the difference is that in the latter form, the string can contain double-quote characters and new line characters without having to be "escaped" so one can write multi-line `String`'s directly; `String`'s have all the same "escape" capabilities for representing individual `Char`'s within the `String` as for the `Char` Type.
8. Tuple literals are just any Type's of literals inside round brackets separated by one or two comma characters such as `(42, "hello")`.
9. Record literals are just instances of a given record Type with literals assigned to its fields written either in its "exact long" form or using the `type alias` Shortcut form as in `{ x = 42.0, y = 53.7 }` or if it had a `type alias Shortcut` assigned to it as `Shortcut 42.0 53.7`.

10. List literals can be written as a comma separated list of literals of the same Type inside square brackets or in "cons form" as a sequence of "head" and "rest" separated by the `::` character so for example as `[ 1, 2, 3 ]` or as `1 :: 2 :: 3 :: []` or as any combination thereof such as `1 :: [ 2, 3 ]`.
11. Custom Type literals are just the use of Custom Type Constructor functions with literals assigned to the fields.

Value and Function Declarations

Elm value and function names are declared with names starting with a lower case character followed by the same selection as following characters for Type's: combination of numbers and letters or the `_` underscore character. Everything after the name other than white space and comments but including the `=` symbol must be indented by at least one column beyond the first column of the name. Just as for Type and Type alias declarations, these declarations can be recognized as the name is immediately followed by an `=` symbol, which is then followed in turn by the expression that defines the result of the value or function when it is fully applied (all of its arguments are supplied). For examples, all of the definitions of `main` in the above examples are values, and there are many examples of functions that take arguments to return something.

Elm Expressions

Expressions are similar to functions in that they have some input values/functions that are combined in some way to produce a result, and in fact functions are just a formalized wrapper for expressions as to defining the parameters of the expression which is the function body where inputs that are not parameters make the function a "closure" that captures the state of those non-parameter values at the time the closure is defined. Expressions are the prime building blocks of definitions in that there must be exactly one expression to the right of every `=` or `->` symbol used in a definition, but each expression can also be build of a "tree" of sub expressions separated by physical `( ... )` blocks, by precedence and associativity rules, and by "choice"/conditional expressions which can cause a branching in the expression logic flow.

Choice/Conditional Expressions

Elm has two somewhat similar ways of "splitting" the code path depending on some condition(s), as follows:

1. `if <boolean condition expression> then <expression1> else <expression2` splits the execution depending on the value of the boolean condition expression so that when `True` "expression1" is executed and if `False` "expression2" is executed where the two expression results Type's must be identical. There are no particular indentation rules implied by the use of these keywords other than the entire expression including these keywords must be within the indentation limit of the enclosing block.
2. `case <expression> of` matches the result of the "expression" against a following number of target "pattern expressions" where each target "pattern expression" is followed by a `->` symbol and an expression to form the returned value with all branches returning exactly the same Type. The keywords do not imply any particular indentation other than conforming to the indentation of the enclosing block, but "pattern expressions" must all be indented identically and the `->` and following expressions must both be indented from the matching "pattern expression" by at least one column (although they don't need to be indented from each other).

Compound Expressions

Expressions are formed by the combination of any number of the above two branching expressions, function applications, symbolic operator applications (whether prefix as in with the enclosing round bracket form where the symbol precedes its two arguments or infix form without the round brackets where the operator is used between its first argument and its second) and all symbolic operators have an assigned associativity (meaning whether the left side or right side is evaluated first) and precedence level as per the following table (note that all symbolic operator are binary operators taking two arguments and returning a result).

Operator Expression Precedence and Associativity Rules

precedence	**left associative**	**right associative**
9	>>	<<
8		^
7	* / //	
6	+ −	
5		++ ::
4	== /= < > <= >= *	
3		&&
2		||
0	|>	<|

The above lists the core infix (meaning arguments on either side of the operator) binary (meaning two arguments) operators in descending order of precedence where higher precedence numbers indicate higher precedence.

Function application (by adjacency) is higher priority than all operators and is left-associative: f g h x is interpreted as ((f g) h) x.

- The comparison operators (precedence 4) are non-associative.

Associativity means whether the left or right arguments or neither side of the operator arguments is fully evaluated before application of the operator; thus for comparison operators there is no guarantee that any other than adjacent values or expressions inside round brackets are evaluated before the condition operator is applied.

Definitions Nested in Outer Expressions

There is no limit on the number of levels of inner scopes of local definitions of values and/or functions inside `let <definitions> in` blocks and such local definitions apply to the expression following the `in` keyword with no indentation implied other than as per the indentation limit of the enclosing block. Inner definitions inside the `let ... in` may in turn have their own `let ... in` blocks defined for their definition expressions or there can be further `let ... in` definition blocks defined at any point withing

the expression following the `in` keyword. The lower case names defined in inner definition blocks have access to all names defined in all enclosing scope blocks and no inner name can conflict/be the same as any outer name which it can access (no name shadowing). The only exception for name shadowing/replacement is that names defined in the current module can be the same as names in imported modules; however, when used there must be name qualification applied (the desired module implementation separated by a dot or dots) to solve the ambiguity of which instance is desired. There is currently a bug/problem if the `List a` type is shadowed by a new definition of the `List` Type name as the original `List a` type is not defined in any module (even the core `List` module) and thus there is no qualification that can be applied to refer to it once it is shadowed.

Patterns Used in Expressions

Elm patterns can be used in three places in Elm code, as follows:

1. As a placeholder for function parameters where a complex type can be "destructured" to get access to its sub fields by name; this works for all of tuple, record, lists, and Custom Type's but for the record types the designated field names must match those of the desired record Type and it only works for Custom Types with only one Constructor/variant as if there are more than one the matching is not "total" (covering all the variant possibilities).
2. Pattern matching can be used with exactly the same limitations as above for the "name" of a local definition inside a `let ... in` definition block.
3. The most general use for pattern matching is as the target patterns for a `case ... of` expression where there isn't the Custom Type limitation as the compiler will enforce that all variants are covered by the various pattern matching targets.

When a given total pattern or sub field in a pattern is not used, a `_` underscore character can take its place; when this is done for the entire pattern as a target for a `case ... of` expression, it will always match so if used as the first target pattern, subsequent patterns will never be tested and used, but when used as the last of a range of very large set of targets as in Int matches, it will cover all of the cases not specified by previous target patterns. `Float`'s cannot be used as target patterns because they are not "exact" values with some imprecision implied in their use. One can use both a target pattern and a name for the entire pattern by following the pattern expression with the keyword `as` and a lowercase name, as in the following code snippet:

```
    case (n, _) of
      (0, str) as all -> if str == "thanks" then all
                         else  (42, "whoops")
      _ -> (n, "something")
```

For `List`'s, both cases of empty/`[]` and a `List` node must be covered, as follows where `list` is a list of numbers:

```
    case list of
        [] -> 42
        head :: _ -> head
```

where the `::` pattern matching operator represents a separator between the head element value and the `List a` that forms the tail and may be `[]`/empty or ignored as in the above example.

Advanced Container Manipulation

Various container types in the Elm packages have means to manipulate them in somewhat standard ways as in mapping them with a `map` function, multi-mapping them with a `mapN` function where `N` is typically two to five, and `andThen`'ing them with an `andThen` function. Containers in the "core" standard package that have these facilities are the `Maybe` module/Type which represents something that can fail but there is no further information other than `Nothing` in the failure with success represented by `Just a` where the `a` Type variable represents any (generic) type assigned when it is used, the `Result x a` module/type which is like the `Maybe a` Type except that there can be failure information encoded by its `x` failure type and with `Ok a` on success and `Err x` on failure, and the `Task x a` module/Type with `x` and `a` as for `Result` but representing an **action**/Task to be `perform`'ed `attempt`'ed by the run time (`perform` always succeeds; `attempt` may succeed or fail). Operations using these functions may be chained as in a `Task` may be `map`'ed and `andThen`'ed in a chain to be finally `perform`'ed or `attempt`'ed which produces a `Cmd msg` to be passed to the run time. The following code uses The Elm Architecture (TEA) to run such a chain to measure how long it takes to do some computationally expensive operation:

```
module Main exposing (main)

import Browser
import Html exposing (text)
import Task exposing (..)
import Time exposing (now, posixToMillis)
import Html exposing (text)

expensive : () -> Int
expensive () =
  ...

doit : () -> ( String, Cmd String )
doit () =
  now |> andThen (\ startpsx ->
    succeed (expensive()) |> andThen (\ rslt ->
      now |> map (\ stoppsx ->
        "Result is " ++ String.fromInt rslt ++ " in "
           ++ (posixToMillis stoppsx - posixToMillis startpsx)
           ++ " milliseconds."
      )
    )
  ) |> perform identity

main : Program () String String
main =
  Browser.element
    { init = doit
    , update = \ msg _ -> (msg, Cmd.none)
    , view = \ model -> text model
```

```
    , subscriptions = \ _ -> Sub.none
    }
```

where this time using TEA to form an active program something like the bouncing ball but only using it to perform a chain of `Task` container operations on initialization as in reading the current time as a `Task`, converting the result of the expensive computation to a `Task` with `succeed` as it never fails, then getting the time after with another `now`, and finally converting the two times and the result to a display `String` with a `map` to be passed to the run time system as an executable action with the `perform` converting it to a `Cmd String`. The run time system then produces a call to the `update` function which only transfers the message `String` to the `model String`, which is then passed to the `view` function that passes the `model`String to the `text` function for display on the simple web page. This shows the essence of TEA in a very simple form. The `subscriptions` field is used when there is an external process that can be enabled to independently fire events into our program such as time interval events that are an option not used here from the `Time` module. There is further information about TEA in the Elm Guide as well as further information on the more advanced use of Elm in creating web pages.

The use of containers to form `map` and `andThen` chains as in the above example is equivalent to Haskell's use of Functor's, Applicative's and Monad's although Elm documentation usually never refers to them as such and they don't work quite the same because Elm does not have "type classes"; however, there is no need to be concerned with these concepts as beginning programmers as simply understanding their use as in the above example and the Guide on TEA as well as the documentation on the `Task` module is more than enough until attaining a quite advanced programming level.

The above code is written using a chain of `andThen` and `map` functions to show that Elm code can be written following a Haskell style even through it doesn't have the "syntactic sugar" to avoid having the spacing "grow to the right" when one does it that way. Most Elm programmers would not do it that way but instead would use TEA to change the internal program state as reflected in a "Model" type with many more round trips to the run time system to accomplish the goal. Thus, most Elm programmers would write the above example more as in the following code:

```
module Main exposing (main)

import Browser
import Html exposing (text)
import Task exposing (..)
import Time exposing (Posix, now, posixToMillis)


expensive : () -> Int
expensive () =
    42


type alias Model =
    { start : Int, result : String, answer : String }


initModel : Model
```

```
initModel =
    Model 0 "" "Waiting for calculation..."


type Msg
    = Init
    | StartTime Posix
    | Complete Int
    | StopTime Posix


respondMsg msg model =
    case msg of
        Init ->
            ( model, now |> perform StartTime )

        StartTime psx ->
            ( { model | start = posixToMillis psx }
            , expensive () |> succeed |> perform Complete
            )

        Complete rslt ->
            ( { model | result = String.fromInt rslt }
            , now |> perform StopTime
            )

        StopTime psx ->
            ( let
                intvlstr =
                    posixToMillis psx
                        - model.start
                        |> String.fromInt
              in
              { model
                | answer =
                    "Result is "
                        ++ model.result
                        ++ " in "
                        ++ intvlstr
                        ++ " milliseconds."
              }
            , Cmd.none
            )


main : Program () Model Msg
main =
    Browser.element
        { init = \_ -> ( initModel, succeed () |> perform (always Init) )
        , update = respondMsg
        , view = \model -> text model.answer
        , subscriptions = \_ -> Sub.none
        }
```

The point of this is that there are options, even in the use of TEA, in performing chains of functions with internal state maintained just by how the function nesting is defined or avoiding the function nesting with multiple calls to the run time system.

As a point of interest and looping back to the beginning, the third code example of the bouncing ball near the top of this document is actually using TEA inside the `game` (and the lesser `animation` available from the same `Playground` module/package except that it hides the `update` function entirely and only exposes a simplified `view` function based on defining a `List` of `Shape`'s based on `Time`) just that some of the details have been hidden; however, the `view` function is taking inputs derived from `Msg` and served in `Computer` and the `memory` part of the "Model" and generating a `List` of `Shape`'s that the package is building HTML/SVG from and the unused `update` function is updating its version of a "Model" based on the `Msg` and "Model" types. So if the reader has been playing with and extending that example as they should, they have been using TEA. Another package that deals with graphics in a similar way as the `Playground` module/package used in the third example is the "MacCASOutreach/graphicsvg" package, which while it does much of the same thing, is more up-to-date and extended than the `Playground`. It is recommended that one spend some time with that package, perhaps even before playing with the examples in the Elm IDE although the Elm IDE examples as per the link below is also valuable.

Conclusion

This document describes the syntax and use of the Elm programming language but not its various applications as is explained at a beginner's level in the Elm Guide referenced above. To advance further, a beginning programmer needs to start writing their own code past these examples and those in the Guide as well as thoroughly reading the Guide and the documentation on at least the basic Elm packages as well as following through the examples available on the Ellie-app and the Elm online IDE.