Elm Language Reference

This Elm language reference material has been reverse engineered from the Elm compiler (written in Haskell) version 0.19.1, which is the current compiler as of the time of this writing, as well as extracted from existing Elm documentation in the Elm Guide and the Elm Syntax document (which is incomplete).

Keyword Summary

Keyword	Use
port	port modules
effect, where, command, subscription	effect modules
module, exposing	modules
import, exposing, as	imports
type alias	type aliases
type	custom types
if, then, else	`if ... then ... else` expressions
case, of	`case ... of` expressions
as	applies a name to a full pattern or sub-pattern
infix, left, right, none	used in Kernel packages to define symbol operators

These last keywords can define a new symbol operator, but can only be used from Kernel (sanctioned) package modules; they can assign left/right/none associativity and precedence from 0 to 9 and are used as follows

"main" is not a keyword but there must be value of this name in the Main module with a type of `Program flags model msg` or `Html msg` or `Svg msg` where `flags` must have a definite type that is expected to be delivered by the JavaScript call to "init". The JavaScript generated by the Elm compiler expects to be started by a call from JavaScript as follows: `Elm.Main.init();` when this is a "headless" module (without a graphics interface - no HTML or SVG, although there are still ways to pass "flags" into the application via this mechanism as explained in the Elm Guide) or `Elm.Main.init({ node: document.querySelector('main') });` when the call is embedded in a HTML web page where it passes the `main` HTML element to form the "root" of the HTML tag structure and returns a representation of the entire Elm application, which can be used for such things as "hooking up" port calls to the application as explained in the Elm Guide.

Symbol and Operator Summary

Note that there are no semicolons (";") that have any meaning in the language as putting more than one definition or expression on a line is not allowed and the expression will be terminated with the indentation rule that the current entity ends when the indentation is the same or less than the previous

declaration or definition, which is consistently used throughout the language; also, the need for block delineator symbols such as curly brackets is replaced by this same indentation rule.

Symbol	Use
--	starts single line comment
{-, -}	starts and ends (nested) block comment
{-|	starts documenting comment
(..)	expose all in module or all Constructors in Custom Type
(..)	import all in module or all Constructors in Custom Type
(comma list)	expose specified in module without qualification
(comma list)	import all in module without qualification
(..,..), (..,..,..)	constructor for Tuple Type
(..,..), (..,..,..)	pattern for Tuple Type
(...)	separates a Pattern Match Term in a function definition or destructure
(...)	separates a function parameter in a Type Annotation
(...)	change precedence within expression
{ .., ... }	used to define Record Type including field types
{ .., ... }	constructor for Record Type
{ .., ... }	pattern for Record Type
|	separates a series of Custom Type Constructor Type definitions
|	used in Record update syntax to specify fields to change
:	separates the value or function name from its Type in a Type Annotation
\	defines the start of a lambda definition before its argument variables
->	separates the arguments of a function from each other and the result in a type annotation
->	separates the arguments of a lambda from its result expression
->	separates `case ... of` target pattern from its result expression
=	binds an already defined Type to a Type Alias Name
=	binds an expression to a Record field
=	binds a series of Constructor definitions to a Custom Type Name
=	binds an expression to a value definition
=	binds an expression to a function name definition

Symbol	Use
=	binds an expression to a pattern in a definition
'	pair used to create a `Char` literal
"	pair used to create a `String` literal
"""	pair create `String` literal preserving return, new line, and quote characters
[..., ...]	used to construct a List where all elements have the same Type
[]	used to construct an empty List
... :: ...	used to form a new List combining a head to an existing tail List
... :: ...	used to pattern match the head and rest of a List
^, *, /, //, +, -	arithmetic operators for number's/Float's/Int's
==, /=, <, >, <=, >=	comparison operators for comparable/comappend types
&&,	
++	append operators append and comappend types (List's and String's)
	>, <
	.,
<?>, </>	adds a "?" or a "/" to a URL for "elm/url" package

Operator symbols are all defined in elm packages and are defined as per the following example for a new `(^^^)` exclusive or operator:

```
import Bitwise
infix left 3 (^^^) = Bitwise.xor
```

where the new infix operator `^^^` is defined to have left associativity (meaning that the expression to the left is evaluated before application of the operator) with a precedence of 3 (with 9 as highest priority and higher numbers getting evaluated before higher numbers if there are no brackets with function application the very highest priority) and the expression at the right of the equals sign is the code that actually gets run when the operator is used. Only binary operators can be defined meaning that all defined operations must take two arguments and return a single result.

Language Elements

White Space Syntax

Elm is a "white space designated scope" indentation sensitive language so white space is significant in the text source files for Elm, where "white space" is specified to be "space" characters, return characters (`\r`), and line-feed characters (`\n`) although both line and block comments also qualify as white space separators.

Specifically forbidden in an Elm text source file other than inside line comments, strings, and characters is the tab character (`\t`), which if encountered by the compiler other than in comments and strings will cause an immediate abort of the compilation process with an appropriate error message. As to "scope", all definitions of modules, imports, Type Name definitions, and value and function definitions that start at the first column of a line are of the "global scope" and a given "global scope" definition will end when subsequent indentation is again at the first column where subsequent non white space (and non comment) definitions will form the scope of a new definition. Similarly, "local scope's" start at the first character of a name definition of a local value or local function in a `let ... in` block and end when the next line starts at a **strictly equal to** the column where the name starts where the expression that forms the left hand side expression must entirely be indented from the first column of the definition Name whether the equals symbol (`=`) is on the same line as the Name or not or when the `in` keyword is encountered at whatever indentation level (inside its outer scope).

Elm is actually much more flexible in allowed use of "white space" than other "white space" languages in that the above are almost the only rules: as long as they are indented for the scope they are in, keywords and expressions can start at any position within the above scope rules including on new lines, where expressions can span any number of lines with subsequent lines also starting at any indented position other than for the `case ... of` target patterns where, although the target patterns obey the above rules with all target patterns aligned since they are at the same "scope", the target expressions and the `->` symbol must be indented from the first column of the target and the target expression scope ends when the indentation of the next line is the same or less than that of the current case target, as per the following examples:

```
-- good code
dummy =
  let
    something -- the first definition can be arbitrarily indented
      = True == -- the `=` can be on new line if it exceeds indentation
    result =
      case something of
        False -> --- the first target can be arbitrarily indented
          "False"
        True ->
          "True"
  in result
```

```
-- unaccepted code
dummy =
  let
    something = -- the first definition can be arbitrarily indented
    True -- fails because expression is indented no more than definition
    indented
    result =
      case something of
        False -> --- the first target can be arbitrarily indented
          "False"
         True -> -- because target doesn't align with previous
        "True" -- because expression is indented no more than target
  in result
```

Elm allows any number of spaces as "indentation", but good programming style is consistent in number of spaces used per indentation level with usually two or four spaces chosen by most programmers, with the "elm-format" source code formatter using four spaces although many programmers prefer using two spaces for indentation to keep line lengths short.

Other than the above rules defining new scopes for definitions and `case ... of` targets via indentation, Elm is almost "free form" in that indentation is ignored, although good coding style extends the indentation and formatting rules for readability of code.

Comments

All comments can contain any text characters whatsoever except that only line comments can include tab characters; there are three types of comments in Elm, as follows:

1. Single Line Comments

These comments start anywhere on a line at the `--` symbol and end at the end of the line.

2. Multi Line Block Comments

These comments start anywhere with the `{-` comment block start symbol and continue to the first `-}` comment block end symbol ignoring any single line comment symbols within the block; however, they may be nested, so the comment doesn't end until the number of comment block end symbols matches the number of comment block start symbols encountered to that point. It is an error if the file ends without a matching number of open and close block comment symbols.

3. Document Comments

These are exactly like the above Multi Line Block Comments except they start on the first column with the `{-|` block start symbol instead of the `{-` symbol and cannot be nested. They are expected to contain Markdown formatted text for more formatted printing of built-in documentation by a documentation building program but the contents will only be evaluated if the compilation is given the flag to generate a documentation file. The contents main extension to standard Markdown syntax is that they can contain a `@docs` macro to embed a comma-separated list of Name's of document comments to be embedded where the Name is as per the immediately following definition Name, whether a type, type alias, value, or function and the `@docs` definitions are expected to start at the first column (currently not enforced, although likely will be in a future version) with the comma-separated list of Name's terminated by a new line. There can be at most only one documentation comment assigned to a Name, which is the last one before the name occurs. However, any documentation comment following the module definition line will be included as part of that module definition and is expected to be the documentation comment that contains the `@docs` macros. , meaning that if there is no module definition line with a documentation comment, there will be no documentation even though definitions have documentation comments.

Names

Names of Types must start with a capital letter and names of values and functions must start with a lower case letter; subsequent name characters following the first character may be any alphanumeric character

(upper or lower case) or the underscore character (_); all such names must be unique in that no name can be redefined if that name is already defined in the same or an outer scope (no Name shadowing) other than Name's defined within a given module are allowed to shadow and thus hide Name's from imported `exposing` modules, with the order of definition in the same scope not being important as all Name's defined in a given scope are available to all other names (including Type and Type Alias Names, which are only defined at the global scope) defined in the same scope. In addition, pattern matching representative names for sub fields or argument names for functions (which also may be pattern matches) may use the single underscore character (_) to represent that the matched value is to be ignored but if not `_` and therefore used no pattern match field name (which must start with lower case as they are not Types) may not be the same as any other name in the same or higher scope.

Modules

Every Elm source file is one module where the module Name obeys the above Name rules and should start with an upper case letter which must match the name of the File without the extension; when enclosed in a directory/folder structure within the source "tree", the Name will start with the directory/folder directory tree (where again every directory name should start with a capital letter) separated by `.` characters without separating spaces. So if the source tree is within the directory/folder named "src" and a submodule is in the "src/SubModule/Work" directory/folder chain with a file named DoMe.elm exporting a function called "doMe", then the first non comment or white space line starting at the first column must be as follows:

```
module SubModule.Work.DoMe exposing (doMe)
```

which names the module as per the directory/folder and file name structure and exports the "doMe" function without qualification. The exposing brackets can contain a comma separated list of exported Types, values, and functions, and if it is desired to export Custom Type constructors, they will all be exported by adding `(..)` to the type name as in `Type(..)`. This list is not white space sensitive, so formatting to make it easy to read can be applied. Every application must contain a `main` value of type `Program flags model msg` or `Html msg` or `Svg msg` and normally is expected to be in a module as defined above called "Main" (in file "Main.elm", although this is not checked), so would have a module defined as follows:

```
module Main exposing (main)
```

Modules as specified above are imported by exactly those names when their packages are specified as dependencies in the project JSON file "elm.json" as a direct dependency, including required version number or their directory's/folders are part of the source code tree as defined above.

`port` Modules

Note that a Port Module can only be within an Application project and not a Package project!

An application module definition line that contains the word `port` before the `module` keyword is an Port Module and can contain input and/or output ports as defined in the Elm Guide...

Another way to get one way settings into Elm is the use of flags, which doesn't require a Port Module, as defined in the Elm Guide; note that the "flags" type must generally be specified in a type annotation or by the variable's use in the program as it's type cannot be definitely determined from the JavaScript input so the compiler should issue an error in such cases where the type cannot be definitely determined...

`effect` Modules

Note that a Effect Module can only be within a Package project and not an Application project and can only be defined by "sanctioned" packages within the "elm" or "elm-explorations" ecosystem!

A package module definition line that contains the word `effect` before the `module` keyword is an Effect Module and has some special requirements in defining an Effect Manager, but as it is thought that there are only about ten of these required in all of Elm, this section is not complete as to their details but the definition line will/may also include the keyword `where` to designate the effect manager implementation...

Imports

In order to use named Types, values, and functions that are exported from modules as per the above (other than those [imports that are automatic}(https://package.elm-lang.org/packages/elm/core/latest/)), one needs to import them on non white space or commented line(s) just after the module definition line but before any global Type, value, or function definitions. The following code shows examples of this:

```
-- qualified imports
import List                             -- List.map, List.foldl
import List as L                        -- L.map, L.foldl

-- open imports
import List exposing (..)               -- map, foldl, concat, ...
import List exposing ( map, foldl )     -- map, foldl

import Maybe exposing ( Maybe )         -- Maybe
import Maybe exposing ( Maybe(..) )     -- Maybe, Just, Nothing
```

where "qualified" means that one must use the imported module name or it's "as" synonym separated from the actual imported type, value, or function by a `.` (without surrounding spaces) to "qualify" the use of the import on every use, and "open" imports mean that the name of the import can be used directly without "qualification". Note that the over use of "open" imports can lead to name shadowing conflicts and is to be avoided for the most part. Also, symbolic operators cannot be used by "qualification" and must be "exposed", but that is typically not a problem with shadowing as there are so few of them and their definition is limited to the "sanctioned" repositories within the "elm" and "elm-explorations" ecosystem.

Imports are checked for cyclic dependencies where there should not be a chain of import dependencies back to one of modules in the chain, which is checked at compile time. This document explains how to work around cases that fail due to cyclic import dependencies.

Types

As Elm is statically typed, Type's are a very important consideration for the language and consist of low-level Native (to JavaScript) types that are used the same way in Elm as in JavaScript and all other types which are implemented as constructs built with JavaScript types and/or objects.

Primitive (Native or "Kernel") Types

There is actually only one native type in JavaScript and that is the 64-bit IEEE floating point number represented in Elm as `Float`; however, modern JavaScript engines can also treat the lower order 32-bits contained within the mantissa of a `Float` as a 32-bit `Int` so those are considered as Primitive/Native/"Kernel" as well. In Elm, when `Int`'s are used in Bitwise functions of bitwise and'ing, or'ing, exclusive or'ing, left and right shifting (logical and arithmetic), and bit complementing, the result is always truncated/clipped to 32-bits, but other operations leave the number represented as whatever integer precision the 53-bit mantissa of the 64-bit floating bit representation can sustain.

The Char Type

Unlike JavaScript, Elm distinguishes between single character Char's and possibly zero to any number of character String's as below. `Char`'s in Elm represent UTF-8 characters so are variable length depending on their "code point", with the ASCII character set taking only one byte and thus being compatible with ASCII when limited to that character range.

The String Type

JavaScript has a native string type but it is UTF-16 and isn't exactly as the immutable UTF-8 Elm `String` type so there is a core package, "String", that defines the `String` type to handle the conversion to and fro in the use of the package functions.

The `Bool` Type

The `Bool` type represents boolean values in the Elm language and is actually implemented as an Elm Custom Type with two Constructors of `False` and `True` (note the capitalization) as can be better understood from the Custom Type section below.

The `()` Type

In other functional languages, this type is often called the "Unit" type but in Elm it is unnamed; it represents something like the `void` or `null` argument or return type in some languages and thus may not represent a memory location and size at all (although in Elm it actually does have a memory representation as a Tuple object with no fields). This type is usually used to represent a function argument or return value of "nothing" or a type field placeholder of "nothing" since all functions in Elm (or any pure functional language) must accept at least one argument and return something. Of course, a function that didn't return anything would only be run for its side effect which is impossible in Elm, so functions that have this intention wrap this type in a type that represents a side effect action to be executed by the runtime system, and thus doesn't impact the "pureness" of the functional language.

The `()` type has only one possible value represented by the same symbol of `()` and only has one instance in the case of Elm, which as said does have a memory representation as a JavaScript object with only a tag field.

The `Never` Type

The `Never` Type in Elm represents a Type that can never be used, which sounds useless but it completes the type system. Suppose that a Elm program uses a function that returns a `Html.Html msg` Type but the messaging facility isn't used as the page is static and never changes; in this case it doesn't matter what value is given to `msg`, which could be left as it is or given any Type including the "nothing"/`()` Type but to indicate it is never used, it could be given the `Never` Type which enforces the idea. The `Never` type appears as the argument of the `never` function that can never be applied but if applied would return whatever type completes the Type where it is used; this function, again, completes the Elm Type system in converting the `Never` Type to make the Type where it is used compatible with the same enclosing type where that field may be used. For instance, if one had a web page with `Html.Html Never` elements because it was a static web page but embedded that web page in a dynamic page that did use messages, one could map the inner `Never` message Type to the outer non-`Never` Type using the `never` function even though they never get applied as in the following Elm code:

```
import Html
type Msg = Init | Done
type alias InnerPage = Html.Html Never
type alias OuterPage = Html.Html Msg
-- when one wants to use the inner page in the outer one...
embedHtml : Html Never -> Html Msg
embedHtml staticStuff =
  div []
    [ text "hello"
    , Html.map never staticStuff
    ]
```

where something like the `embedHtml` function can be applied whenever the `InnerPage` Type needs to be made compatible with the `OuterPage` Type even though the `Never` type is never used and the `never` function will never be applied but just makes the Type's compatible.

The Tuple Type

Tuples are containers for two or three possibly different Type's; not zero Type's because `()` represents the "nothing" type as above, not one because `(something)` only represents (redundant) brackets around "something" where something might be a binding name for a value or function or a literal or expression. Therefore (1, 2.0) is a 2-Tuple `(Int, Float)` and `(1, 'a', "abc")` is a 3-Tuple of `(Int, Char, String)`; higher numbers of Tuple fields are not allowed as they don't really make sense as the main difference as compared to a Record is that the fields don't have names but can only be accessed by positional pattern matching. The surrounding brackets can be considered to be the "constructor" when creating new tuples this way.

One can nest tuples in other tuples to contain more values, but "drilling down" by pattern matching to obtain the inner values may not be the most efficient way to do this and again, using a Record is likely the better way to do this.

Every Tuple type with different types of contents is a separate Type, but there may be many instances of values of that Type.

The Record Type

Records are like Tuples with field names but may have zero or more named fields with each field of different Type's and, in Elm, have some nice update syntax when creating a new copy changing only a few fields; however, they have some limitations when extracting fields by pattern matching. Anonymous Record Type's can be created by the following syntax:

```
r = { i = 2, j = 2 } -- r is of type { i : Int, j : Int }
```

However, records are often used with a Type Alias, which then automatically creates a constructor function of the name of the Type Alias as in the following example:

```
type alias TestRec = { i : Int, f : Float }
testr = TestRec 3 7 -- creates a TestRec as above
```

Unfortunately, the current Elm compiler does not continue to creation of "constructor" functions for Type Alias's that are synonyms for other Record Type Aliases, so the following does not create a "OtherTestRec" record constructor:

```
type alias OtherTestRec = TestRec
otr = OtherTestRec 7 42 -- no constructor of this name!
```

Fields of a Record can be accessed by using the field names as "dot suffixes" (without surrounding spaces) or using the field names with a preceeding dot (with no space between the dot and name) as a access function, as in the following example for the above `testr` binding value:

```
a = testr.i
b = .i testr -- a and b will both be 3
```

Elm also has another Record Type feature called "extensible record syntax" (which should really be called "flexible record syntax") that is rarely used externally but can be understood by giving the above `.i` type signature as follows:

```
.i : { r | i : a } -> a -- more generally for any type `a`
```

which says that `.i` takes as an argument any record type `r` containing an `i` field and returns the contents of that field as the type it holds. Other than automatic use in Record Type Accessor functions as shown, one can write their own field accessor functions for more than one field as follows:

```

```

```
testGrab2 : { v | i : a, j : b } -> (a, b)
testGrab2 r = (r.i, r.j) -- or can pattern match:
testGrab2 {i, j} = (i, j) -- since `r` only has the two fields
```

which will accept an argument of any Record Type that contains the indicated fields and can manipulate only those fields even though the argument parameter Record can have more fields, although this ability is now discouraged by the Elm developers for new code. Pattern matching the Record Type is further expanded later

Records can have fields that contain other records, so one can build quite complex data structures only using records.

Elm records have an "Record update" syntax that makes them preferable when producing changed copies that only have a few fields changed: as in the following example:

```
type alias Test = { i : Int, f : Float }
test0 = Test 2 3
test1 = { test0 | f = 5 } -- => same as Test 2 5
```

where the "Record update syntax used to bind a value to `test1` makes a copy of `test0` with only the `f` field changed.

Each Record even with the same field names and Types is a distinct Record Type and each have accessor functions that may have the same name. Theoretically, the empty Record, `{}` need have no memory representation and if it has a memory representation as in Elm currently, could have just one instance but the current Elm compiler isn't that sophisticated.

Some of the above information is also provided in the Elm documentation...

Custom Types

One of the most powerful persistent basic data structures in Elm is the Custom Type, which can be used to build enumerations with many constructor fields with each containing no data or an Int enum index, as single constructor field containers, or as sum types/ADT's (or tagged unions) for complex functional programming uses. Since fields can contain any other Type including Tuples,Record's, or other (or the same) Custom Type's, very complex data structures can be built using them, and they are generally the basis for further extended persistent data structures such as List's, Set's, Dict's, Tree's etc.

Custom Data Types are defined at the global level with the keyword `type` and one or more Constructor Name's (which must be capitalized) each of which can have zero or more data fields of any Type, and data is always extracted by pattern matching as explained here.

Type Aliases

Type Aliases are created at the global level using the `type alias` keyword where the right hand side is an already defined Type which may be a Tuple Type `( ... )` with field types in a comma separated list (two or

three), or a Record type as defined in a pair of `{ ... }` brackets around field type definitions as explained in the Guide and this document explains and walks through some remedies for recursive Type Aliases.

Type Variables

Complex types defined as Type Alias's and Custom Types can be containers for other Type's which can be parameterized for as many contained Type's as necessary. For instance, a generic List must be parameterized by the constant type of its list elements, if any, and a `Dict` lookup type would be parameters by the two Type's of its keys and its values. These Type Variables are just lower case simple names that represent the value of the Type when the container type is instantiated and such Type Variables must be common to both the left hand side of the Type definition and the right hand side showing where the instantiated types are used.

Note that current Elm does not support the idea of Higher Kinded Type's (HKT's) which would allow the manipulation (say) of the List Type using only the `List` without its type parameter(s); this is mostly useful for higher order functional programming such as building "type classes" which current Elm does not support (probably in order to keep it simple).

Type Constraints

In Elm, there are defined a few Type Constraints to be used in the place of a group of defined Type's as follows: `number` represents an Int or a Float; `appendable` represents a List or a String; `comparable` represents any of `Int`, `Float`, `Char`, `String`, and List's/Tuple's of `comparable` Type's; `compappend` permits `String` and `List comparable`'s. When used as a Type variable to represent one of its choice of types of the left hand side, it must represent that same type in all locations where it is used on the right hand side other than if it is desired to designate several places that a given context is to be used with a different type value for each, one may append a number to the name for instances which are to be equivalent as in the following example:

```
test : number -> number1 -> number
test v0 v1 = v0 -- and also might include v1 in the result
```

This indicates that this function "test" can take any of the following combinations:

1. take an Int and an Int to produce an Int
2. take an Int and an Float to produce an Int (if that is somehow possible)
3. take an Float and an Int to produce an Float (if possible)
4. take an Float and an Float to produce an Float

The trouble with the above is that although it can be defined, it is difficult to use practically because all of Elm's current number conversion functions are written to convert a definite type such as `Int -> Float` or `Float -> Int` and thus can't be used generically as required here.

Other Core Package Defined Types

All Elm Type's are actually defined in the Core Package other than the Tuple Type, Record Type, and List Type (which require compiler "magic" to work), and for Primitive Type's that need Native/"Kernel" code to make

them work that are defined as in the following example for `Float`:

```
type Float = Float -- NOTE: The compiler provides the real implementation.
```

or as a Custom Type with one Constructor which is the same Name as the Name of the Type; thus, all of the specified Primitive Type's above are actually defined in the Core "Basics" Package, namely "Int", "Float", "Bool" (a Custom Type with only two Constructors, "False" and "True", neither containing any data), and "Never" (a type which the compiler prevents from ever being used because it is an infinitely recursive global Type).

Although the current Core Package contains "List" and "Tuple" modules, these only contain a set of utility functions for the case of "List" and only for 2-Tuple's in the case of "Tuple" as the compiler needs to hand the construction and pattern matching of these Type's by "compiler magic".

Specialized Type's also have their own module that at least exports the Type, may export the Type Constructors, and may export all of the values and functions related to low-level use of the Type; these include the "Char" and "String" modules as well as the "Maybe" and "Result" modules (which last two include exporting their Constructors) and parts of these modules are automatically imported into every module.

Other specialized modules for very useful persistent data structures that aren't automatically imported are "Set", "Dict", and "Array" (a tree-based array representation, not a linear array), for which the Constructors are not exported.

Finally, the Core Package contains a "Process" Type of "Id" (related to use in concurrency but which is unusable in the current implementation), some "PlatForm" types as in "ProcessId", "Task", "Router", with only "Program", "Platform.Cmd.Cmd", and "Platform.Sub.Sub", all exported without Constructors with only these last three Type's automatically imported to a project; all of these are related to use in running a Elm Application as documented in the Elm Guide.

Literals

Elm Literals may take the following forms:

- A bare number without a decimal or exponent could be either an Int or a Float; to force to be an Int when it isn't inferred, a Type Annotation must be used or a function the returns an Int is used such as `floor`. To force it to be a Float, the easiest way is to add a decimal and a zero for the fractional part (which may not be an exact representation depending on the value).
- Int literals have a maximum range of -2^31 to 2^32-1 and Float literals have the range and precision of an IEEE double precision floating point number.
- Char literals are inside a pair of right appostrophes/single quotes and may include escaped characters as per the following table, where escaped characters are represented by a single '/' character followed by one of the following characters to represent a given unicode code point:
    1. '/' meaning a double back slash which represents a single back slash in the Char.
    2. 'r' to represent a carriage return character.

3. 'n' to represent a new line character.
4. 't' to represent a tab character (forbidden except in line comments the source code, but not in characters and strings).
5. '"' to represent a single double quotation character.
6. "'" to represent a single back tick (apostrophe) character.
7. 'u' immediately followed by four to six hexidecimal digits (upper or lower case) enclosed in `{..}` characters representing unicode code points from 0x000000 to 0x10FFFF.

- String literals are inside a pair of double quotes and may include escaped characters and/or Unicode code points as above. String literals may also be inside a pair of sets of adjacent triple quote characters, in which case the carriage return and linefeed characters inside the string will be preserved meaning one can define very long multi-line `String`'s this way. Since indentation and tabs don't have a meaning, the "triple-quoted `String` contents can be defined right from the left column and can include tab characters just as can regular `String`'s.
- Bool literals are either the values of `False` or `True`.
- "nothing" literals are the value `()`.
- The `Never` Type has no literal value as it can never be used.
- Tuple literals are inside a matching pair of round brackets with element assignments separated by commas.
- Record literals are inside a matching pair of curly brackets with field assignments using the '=' character comma separated from each other. If a record has a type alias which is immediately assigned to a given record, the type alias can be used as a "contructor function" without requiring curly brackets or field assignments where fields are assigned according to their position.
- List literals are inside a matching pair of square brackets where elements (which must all be of the same Type) are comma separated, or consist of an empty List with no elements.
- Literals for other complex Type's built up from the available Type's are created just by using their constructors as a "constructor function".

Values

In Elm, a value definition is simply a name bound (by a `=` symbol) to the immutable returned result of an expression including literal constants with operator applications, function applications, binding Name(s) used to build the expression along with round brackets which may be used to force the precedence of applications (function application is always at the top zero level and are left associative). There are special conditions for non-global values not being cyclic as to their definition as explained in the local definitions section.

Functions

Functions in Elm are very similar to Elm values except that they have one or more arguments following the function binding name and the result of the function is deferred to when the function is called with value arguments (which may even be functions that can also be considered as values) for all of its arguments as

applied to its result expression. So a value is an immediately computed binding, but a function is a binding that represents the calculation of the function expression when called with values for all of its arguments.

Elm functions are curried and may be partially applied, meaning that if a function has many arguments and only something less than the number of arguments is applied, the return value is a new function with the remainder of the arguments to produce the return value.

One may also have functions that are defined without names (or anonymous functions), often called lambdas, such as the following:

```
testFunc = \ x y -> x + y
testFunc 2 3 -- => 5 as called here
```

where everything to the right of the `=` is the lambda/anonymous function which is bound to the value named "testFunc" and called via the bound name.

Finally, if functions, whether anonymous or named, use bindings from outside their internal scope which includes their argument names, they are called "closures" and "capture state from outside their scope, as in the following:

```
testClosure () =
  let x = 42
      clsr y = x + y
  in clsr 7
testClosure () -- => 49
```

where `x` is the captured external value of 42 and `clsr` represents the closure that captured it.

Elm also supports recursive functions (functions that call themselves) as the only way that one would be able to do iteration without mutation, which Elm as a pure functional language doesn't have, as follows:

```
squaresTo n =
  let loopi i lst =
        if i < 1 then lst
        else loopi (i - 1) ((i * i) :: lst)
  in loopi n []
squaresTo 10 -- => [ 1, 4, 9, 16, 25, 36, 49, 64, 81 100 ]
```

where the work is done by the `loopi` inner function calling itself. This is a tail-call recursive function because the call back to itself is the "tail" of computation and no value gets passed back to be handled recursively through stack operations; thus the compiler can turn the `loopi` function body into an imperative loop "under the covers" and not build stack no matter the number of loops.

Local Definitions of Values and Functions

The use of the "let ... in" keywords allows local definition of values and functions as defined above where the names (one or many) to be defined in a given `let ... in` block represented by the `...` must be indented from the `let` and aligned with each other and the expressions bound to each must never start at at this alignment column position or less. The scope of such defined variables or functions is all of the names defined following such a `let` until the next `in` (and may be nested) and local values and functions defined apply to the expression following the closing `in`.

Functions can always be recursive and recursively defined, but non global values can not be cyclic where the name of the value is used on the right hand side if the definition no matter how deep the chain of computations that depend on the value, which is checked at compile time, as in the following examples:

```
type Stream a = Stream a (() -> Stream a)
globalTest =  Stream 42 (\ _ -> globalTest) -- okay
let
  localTest =
    Stream 42 (\ _ ->   localTest) -- value recursion error
in localTest
```

where the global case forms an infinite stream of the type a in head position but the second doesn't pass compilation because of the cycle in the definition. Older versions of Elm allowed recursive use of values as long as the reference was inside the body of a function, but even that is no longer allowed in the interests of allowing for a reference counted type of memory management rather than JavaScript's automatic Garbage Collection and this was the easiest way to allow that. Cyclic value definitions are rarely required and there are often better algorithms for avoiding even these few requirements. There may be more complex ways of again permitting them, but the effort would only be worth it if there was a more frequent requirement for them.

Type Annotations

Type Annotations are placed before and at the same indentation level of the name binding for a value or function definition on the next preceding line which is not white space or comment. They use the symbol `:` to separate the given name from the Type Annotation, and for functions use the `->` symbol to separate each argument Type and the final result Type, as in the following example:

```
hyptenuse : Float -> Float -> Float
hypotenuse x y = sqrt (x^2 + y^2)
```

For function arguments that are functions (nested to any level), the function parameter must be surrounded by round brackets/parenthesis to separate the sub-function from the outer function as in the following example for the `map` function over list's:

```
map : (a -> b) -> List a -> List b
```

where the first argument to `map` is a function taking a Type `a` and returning a Type `b` identified by Type variables, which shows it is the converter function converting the input `List a` to `List b`.

The compiler will warn when any global value or function definition lacks a Type Annotation. This document explains workarounds for Type annotation problems.

Operators

In Elm (and Haskell and F#), a symbolic operator is just a function when used with its enclosing round brackets, but can be "infix" when defined as such when used without the brackets. When `infix` is used, the operator is given left/right/none associativity and a precedence level, with higher precedence numbers having higher precedence in order of execution and with all precedence levels lower than function application which is left associative. All defined symbolic operators must be binary operators (two arguments).

Core Package operators: Basics (always available): Math operators as applied to the `number`'s type except where noted:

precedence	operators	associativity	description
8	^	right	raise the left to the right power
7	*	left	multiply the two values
7	//	left	divide the first by the second (Int)
7	/	left	divide the first by the second (Float)
6	+, -	left	add or subtract second from first

Comparison Operators (for all `comparable` or `compappend` types):

precedence	operators	associativity	description
4	==	non	test for equality to the second
4	/=	non	test for non-equality to the second
4	<	non	test for first less than the second
4	>	non	test for first greater than the second
4	<=	non	test for first less or equal to the second
4	>=	non	test for first greater or equal to the second

Bool operators (for two Bool types):

precedence	operators	associativity	description
2	&&	right	result of logical "and" of first and second
2	||	right	result of logical "or" of first and second

Append operator (for appendable or compappend types):

precedence	operators	associativity	description
5	++	right	appending of first and second

Function pipeline and composition operators (for two single argument functions):

precedence	operators	associativity	description
9	<|	right	passes forced right to left function
9	|>	left	passes forced left to right function
0	<<	left	passes result of right function to left function
0	>>	right	passes result of left function to right function

List operator (automatically imported where head must be the same time as the contained type in the List):

precedence	operators	associativity	description
5	::	right	adds left value to head of right `List`

Normally it is discouraged to use operators in Packages, but the elm team have broken this rule in a couple of "parser" Package's to make the syntax more concise as follows:

"elm/parser" Package operators (works on two package `Parser`'s to produce a `Parser` type):

precedence	operators	associativity	description
6	|.	left	ignores second even if parsed correctly
5	|=	left	keeps correctly parsed second added to first

"elm/url" Url.Parser module operators (applies to `Url.Parser.Parser` types):

precedence	operators	associativity	description
8	<?>	left	adds question to URL path Parser
7	</>	left	adds slash to URL path Parser

Since Elm only allows the definition of new operators in the "Kernel" packages as controlled by the key Elm development team, it is unlikely that Elm will have operators other than the above.

Expressions

Pure functional languages like Elm don't have program statements but only definitions of values and functions that return expression applications (immediate or deferred, respectively), which may be simple or compound. Expressions may be as simple as just a literal or as complex to any level by combining function applications and infix operator applications to literals or the results of sub expressions as arguments with round brackets used to modify default associativity and precedence.

`if ... then ... else ...` Expressions

These three keywords are an expression where depending on the first expression being True or False, the result is either the second expression or the third expression, with both of the latter expression of the exactly the same type (compiler enforced).

`case ... of` Expressions

These two keywords make a selection based on the result of the first expression to pass to a chain of target literals or pattern matches to return (using the `->`) the result on the right hand side of the first match as explained at the end of the Conditionals section here; the return expressions must all be of exactly the same type for a given `case` expression (compiler enforced).

Changing or Asserting Expression Precedence

Calling functions and using operators have a defined default precedence or ordering of operations; however, to confirm and/or change the default precedence, one may use brackets (`(...)`) or nested brackets around sub expressions in order to obtain the desired order of operations. For example, the following expression will perform the addition before the multiplications even though the default is the reverse:

```
2 * (3 + 4) * 5 -- => 70 not 120 as without the brackets
```

Pattern Matching

Pattern Matching, which is called destructuring in some languages, can be used in three places as follows: in the arguments for functions to determine the field values of complex types (only if there is only one constructor as otherwise it is not **Total** meaning all cases covered), as a `let` left hand side with the same condition as to single Constructor, or its most common use as a `case ... of` target where all Constructor possibilities can be covered by using multiple pattern matches in multiple case targets. Any given pattern can have an `as` suffix to provide a name for the entire pattern "term" if it matches. Note that a set of round brackets/parentesis with nothing inside is a valid pattern match for the "Unit" Type and that also that a set of round brackets/parentesis surrounding a single Pattern "term" only separates out the Pattern "term" (optionally including an `as` alias) as a valid Pattern "term"; neither of these are cases of "Tuple Pattern's" as per the following category.

Tuple Pattern Matching

Tuple pattern matching works in all cases as there is only one "constructor" such as the following example:

```
let (_, y) as t = (2, 3) -- note the whole tuple is t
in y -- => 3.0
```

where the left pair of brackets are the Pattern Matching defining a Tuple and the right pair of brackets are the tuple "constructor". Note that we can use `_` for fields to be ignored

Record Pattern Matching

Record Pattern Matching works the same as Tuple Pattern Matching except the pattern field names must exactly match all of the Record field names, which can lead to some variable name clashing, as in the following example; also, the record pattern match must exactly match the field names so one can't use `_` to ignore some fields:

```
type alias TestR = { i : Int, f : Float }
let
  { i, f } as t = TestR 2 3.0 -- note the whole record is t
  { i, f } = TestR 4 5.0 -- NAME CLASH of i and f!!!
end (i, f)
```

which won't work because the second pattern match shadows the `i` and `f` bindings. This is only moderately useful in limited cases where all fields must be extracted just once as pattern matching a function argument...

When one wants to pattern match only on some of the Record Type field names, one can use the "extensible record syntax" (which should really by called "flexible record syntax") as explained previously.

When Record pattern matching isn't convenient due to the restrictions above, it is better to not use pattern matching but just extract the fields through the use of the "dot" syntax or functions.

Custom Type Pattern Matching

When one must use pattern matching for more fields than the two or three offered by Tuple's, a better use case is using a Custom Type as follows:

```
type TestCT = TestCT Int Float -- single constructor
let
  TestCT i _ as t0 = TestCT 2 3.0 -- whole type is t0
  TestCT i1 f as t1 = TestCT 4 5.0 -- whole type is t1
in (i, i1, f) -- => (2, 4, 5) : ( Int, Int, Float )
```

where one is free to use different names for the different fields for the different pattern matches and free to use `_` to ignore some fields.

An example of use of a Multiple Constructor Custom Type to create the `Maybe a` Type with the `withDefault` function with a `case ... of` expression, as follows:

```
type Maybe a = Nothing | Just a -- two Constructors
withDefault : a -> Maybe a -> a
withDefault dv mybe = -- can't use patter here
  case mybe of -- use pattern match as case targets...
    Just r -> r -- if Just, return data contents
    Nothing -> dv -- TOTAL coverage, Nothing has no data
```

List Pattern Matching

Elm `List`'s may also be used in pattern matching where `[]` is the pattern for an empty `List` and `a::ras` is the pattern for a non-empty `List` where `a` will be bound to the head element of the matched list and `ras` will be bound to the tail `List` of the matched `List` if the pattern matches where an optional `as` name can be used to match the whole `List` on a match.

WebGL Shaders

Modern languages often support embedding describing graphics textures and rendering 2D and 3D shapes using these textures as Domain Specific Languages (DSL's) in their syntax to direct commands to the graphics CPU; Elm does this for its web pages by using WebGL specifications (Web Graphics Language) to embed C language-like syntax shader directives inside the `[glsl| ... |]` expression where the block of code inside the delimiters is OpenGL Shading Language (GLSL) that is compiled to instruct the Graphics Processing Unit (GPU) on the textures used and how to render the graphics. This shader language syntax is not directly part of the Elm syntax and must be learned from various documentation available from the Internet such as this one.

Appendix

Project JSON Formats

For Applications:

```
{
    "type": "application",
    "source-directories": [
        "src"
    ],
    "elm-version": "0.19.1",
    "dependencies": {
        "direct": {
            "elm/browser": "1.0.0",
            "elm/core": "1.0.0",
            "elm/html": "1.0.0",
            "elm/json": "1.0.0"
        },
        "indirect": {
            "elm/time": "1.0.0",
            "elm/url": "1.0.0",
            "elm/virtual-dom": "1.0.0"
        }
    },
    "test-dependencies": {
        "direct": {},
        "indirect": {}
    }
}
```

as documented.

For Packages:

```
{
    "type": "package",
    "name": "elm/json",
    "summary": "Encode and decode JSON values",
    "license": "BSD-3-Clause",
    "version": "1.0.0",
    "exposed-modules": [
        "Json.Decode",
        "Json.Encode"
    ],
    "elm-version": "0.19.0 <= v < 0.20.0",
    "dependencies": {
        "elm/core": "1.0.0 <= v < 2.0.0"
    },
    "test-dependencies": {}
}
```

as documented.

Package dependencies may be cyclic where there is a chain of dependencies back to one of packages in the chain as these lists of dependencies for all packages is merely merged to ensure that the packages are installed and incrementally compiled to provide the compilation artifacts in the package cache directory/folder.