Functional Programming in Elm for Imperative Programmers

- Functional Programming in Elm for Imperative Programmers
  - Introduction
  - The Basics of a Functional Languages
  - Misconceptions about Functional Languages
  - Pure Functional Programming is "Safe"
  - Learning Elm versus Haskell
  - About Significant White Space Scoping
  - How to Not Require Mutation
  - What can One See from the Examples
  - The Details of Elm Syntax
  - Advanced Concepts
  - Conclusion

Introduction

There is a great roadblock to the advancement of "pure" functional programming languages because the programming paradigms used by the majority of programmers who are imperative programmers (using languages such as Python, C/C++, Rust, Java/Kotlin, C#, etc.) are not functional paradigms and find such paradigms difficult to grasp. This is due many preconceptions and misconceptions, as will be explained in this document. Functional programming is not "hard" just "different"!

The Basics of a Functional Languages

There is only one condition that defines a "pure" functional language (of which there are few, being Haskell with its derivatives such as Purescript and Elm): it is a language that doesn't allow any "side-effects" or functions that don't return anything and are applied just for their effect on the "real world" including mutation/changing of values/printing/file output/obtaining "real world" inputs such as time/random values/file contents/etc. This means that a "pure" functional language's functions always return the same thing (which they must always return something) given the same function argument(s), which in turn makes "pure" functions "referentially transparent" meaning that a function together with the application of its arguments can be replaced just by the functions result for those arguments.

Just having Higher Order Functions (HOF's) such as generalized map, filter, and fold functions and "lambda's" (anonymous functions that can be used as first class values) doesn't make a functional language as they are just implementation details; in fact, HOF's are often/almost always implemented by a library package even for functional languages (for both Haskell and Elm).

Misconceptions about Functional Languages

One of the biggest misconceptions about functional languages is that they must needs be slower than imperative languages. This is not a general truth, as demonstrated by that well-written Haskell code can often be just as fast as well-written C/C++/Rust code with the cases where it is slower usually just due to implementation details such as the implementation of Garbage Collection memory management or the lack

of certain optimizations such as SIMD. Elm code, which compiles to JavaScript as its "back-end", is often slower than raw JavaScript code written to take bast advantage of modern JavaScript engines in web browsers, but that is just due to the current Elm compiler not being very efficient as to its optimizations and there is nothing stopping a new version from improving on this.

Pure Functional Programming is "Safe"

Over the past several years there has been increasing concern with building type and memory safety into programming; the programming language, Rust, distinguishes itself by providing a means to ensure this "safety" in using fairly complex language syntax and constructs that ensure guarantees about variable lifetimes and when they are allowed to be mutated/changed as well as providing a memory management model that doesn't require "Garbage Collection" (which means the runtime at some indefinite time doing a scan for unused memory allocations) so as to be able to free them safely and deterministically (at a known time). However, "pure" functional programming is magic in that none of the complex language constructs are necessary as to tracking lifetimes of "borrows" as all values are immutable/never change (other than some "kind of safe" ways of mutating the contents of arrays and the target of references in Haskell which may also be "unsafe" as to bounds checking and therefore only to be used by someone who knows what they are doing - there are standard forms of using these which are generally safe and easy to use). So all of the complexity of these extra constructs are unnecessary (even Rust has "unsafe" ways of bypassing this protection by experts when necessary). So "pure" functional languages such as Haskell and Elm are "safe" by their very nature, yet generally without complex syntax to ensure this.

As to "memory safety" in the sense that a given (immutable/unchangeable) value must be valid when accessed but its memory allocation should be freed when the binding to that value is no longer used, current functional languages pretty much exclusively use Garbage Collection (GC) runtime operations to do this (Elm by virtue of that its target JavaScript uses GC), but that again is just an implementation detail: there is no reason that Haskell (for instance) could not use memory management based on smart (elided counting when redundant) reference counting on memory allocations as long as it was "behind the scenes" for the programmer such that memory was made available on binding a value to a value and the value's memory was freed as soon as the binding was no longer used, which could be determined at compile time by data flow escape analysis. This would also be true for Elm given a different "back-end" (say a Web Assembly one) where this smart reference counting would also make sense.

So whether a programming language has GC or not makes no difference to the class of programming paradigms it falls under; while all current functional languages whether "pure" or not have GC, that is just an implementation detail of the platform on which they run (Clojure and Scala on JVM, F# on DotNet) or their history (LISP and its derivatives such as Scheme and Racket, OCaml and its predecessors, Haskell), which history was determined by necessity as in "smart" compilers weren't perhaps possible at the time the preceding languages first appeared. Future "pure" functional languages may well no longer use GC.

Learning Elm versus Haskell

While Haskell may be the "ultimate" in "pure" functional languages as to features, it is not the best "pure" functional language to learn for a beginner to functional programming for the following reasons:

1. It is an experimental research language so not the most stable as to how features may be used over versions as a typical Haskell programmer will almost always use "Extensions" beyond the Haskell specification.

2. It is a "lazy"/"non-strict" language meaning that every value is not computed immediately when defined but only when it is needed to be used as a value and then it is computed only the first time it is used. While this allows certain types of mathematic and algorithmic expressions that would not be possible without laziness, a non-lazy language is still able to express these by using **optional** laziness where needed. The problem with "always on" laziness is that it makes the program data flow hard to reason about, especially for beginners, where one can build huge stacks of deferred evaluations without realizing it.
3. Its syntax is quite complex and a bit obscure in providing many ways of accomplishing a given goal.

Compare this to Elm as follows:

1. While Elm is still not a "stable" language as in guaranteeing that features will never change, it has not been updated for almost four years as of the time of this writing and isn't likely to change much and if it does it will be only to add one or two features that likely won't affect the syntax much if at all.
2. Elm is non-lazy by default with some ways of deferring computations optionally.
3. Elm has a "stripped down" syntax from those of primarily Haskell but also deriving some syntax from F#. Thus, there are only about a dozen keywords one must learn and they are always used in the same way; there are only about twenty non-alphanumeric symbols used and they are very similar to their use in other languages (addition, subtraction, multiplication, division symbols, etc.).

About Significant White Space Scoping

Imperative programmers who are accustomed to the "curly's" or some sort of "begin/end" scope boundary markers (or Julia's non-symmetric "end" marking the end of a scope) often complain about lack of these in "significant white space" languages such as Elm. To keep it simple, Elm only has three indentation rules, as follows (where "white space" includes all forms of comments: line, block, and documentation):

1. All type definitions are at the "global" level and thus the `type` keyword (or `type alias` keyword pair) start at column one and the type definition ends when there is something other than white space in a following line on column one.
2. As for type definitions above, for function or value definitions the definition ends when the indentation level falls back to the level of the name of the definition or less. This is consistent for global value/function definitions where the name is on column one or for local definitions inside `let...in` blocks where the name will be at some arbitrary column higher than the indentation level of the first letter of `let`.
3. For a series of definitions, each new name definition in the series must be at the same indentation as the rest of the name definitions in the series and the series ends at the `in` keyword with the `in` keyword being looked for if the indentation is less than the series name definition indentation level. This is consistent with `case...of` targets where the target patterns take the place of the "name" definition that all must have the same alignment but with any alignment as long as it is to the right of the next level outer definition and all of the code expressing what to do when the pattern is matched as in the `->` and the following expression must be to the right of the target pattern to be classed as part of that expression.

The following demonstrates this:

```
type Whatever = What -- first type definition
name = "Whatsit" -- value definition
```

```
type alias Whoever = Whatever -- new type definition, and order
doit() = -- global function definition
    let x = 42
        y =  7
--            z = 55 -- error: `z` is too far right
         in -- can be anywhere
 case x of -- can be anywhere
 3
  ->
  (1, 2) -- must be to the right of `3`
 _ -- match everything else
  -> - -- must be to the right of `_`
  (x, y)
 -- end of `case` target block
 |> do other something with tuple -- back to case indent or less
-- end of `doit` block
```

Now, using good styling practice, one would probably never write code as ugly as the above, which just shows how flexible the indentation rules are. As to definitely showing the end of a definition block or a `case` target block, feel free to put in comments showing what block is ended just as many "curly" programmers do to show what block the closing "curly" represents the termination of that block as has been done in the above example.

How to Not Require Mutation

When imperative programmers first encounter non-pure functional languages that allow mutation, the tendency is to treat it just as a imperative language by using mutation to implement loops, etc., thereby gaining none of the functional advantages and likely losing in performance or code elegance and/or convenience in that these features aren't intended for common use. An example is that while mutation can be used to implement `while` and `for` loops in F#, there is no `break` or `continue` statements where an imperative programmer commonly uses these. However, `continue` and `break` are easy to implement using recursive function call looping as will be explained.

The only way to be able to loop without requiring mutation in a pure functional language as is required by a imperative language `for` or `while` loop is by recursive function calls. An example is in order here, as follows for the (in)famous Fibonacci number calculation where each element in the series is the sum of the preceding two with the first two zero and one:

```
nthFib n =
  let loop i last current
        if i > n then current
        else loop (i + 1) current (last + current)
  if n <= 2 then min 0 (n - 1) -- allow for n < 1!
  else loop 2 1 1
```

whose explanation in words is quite simple: for the first and second numbers just output 0 and 1, respectively; for succeeding numbers, recursively loop determining each current value as the sum of the two last until the index number exceeds the desired index in which case output the current value. There is

no need to make a value mutable here although all of the function arguments are being assigned different values for each recursive loop.

In actual implementations of recursive functions as expressed above, the stack is not being built on every recursive call because generally the compiler will be turning the inner loop into a real loop in machine code with the three values actually mutated there, but the programmer never sees this. However, that does bring up something that functional programmers do need to think about: The above code can be turned into a loop because the recursive call is in "Tail Call Position" (TCP) meaning that the caller calls it as the last "tail" thing it does and its code is not dependent on the returned result for further computations, thus it can be turned into a simple loop.

The following implementation of the Fibonacci function does not use the recursive call in TCP and thus builds stack on every loop:

```
nthFib n =
  if n <= 2 then min 0 (n - 1) -- allows for n < 1!
  else nthFib (n - 2) + nthFib (n - 1)
```

This incredibly inefficient but frequently implemented recursive implementation does not call using TCP because each iteration is dependent on the results of the next so it is building a stack of pending result calls and, what is worse, each value calculation is dependent on a new stack of calls so instead of `O(n)` complexity as for the first TCP implementation, this as `O(n!)` (the factorial of n)! But it does clearly demonstrate that functional programmers need to keep in mind TCP. Of course, there are ways to minimize the bad effects of the above non-TCP calls in memoizing the results of each sub call so that the complex nesting is only needed once and avoiding the factorial complexity, but the point is made.

Many functional programmers will immediately try to use HOF's, especially involving lists, to solve the above problem, which might be implemented in Haskell as follows:

```
nthFib n =
  let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
  in fibs !! (n - 1)
```

where it obtains the nth value of the Fibonacci sequence (the `!!` index of) although that isn't exactly able to be implemented in Elm because Elm `List`'s are not lazy as are Haskell's lists. We can implement the algorithm using self-implemented lazy Co-Inductive Stream's (CIS's) as follows but as those are not memoizing, we run into a huge performance problem:

```
type CIS a = CIS a (() -> CIS a) -- infinite
tailCIS (CIS _ tail) = tail()
nthCIS n (CIS hd tail) =
  if n <= 1 then hd else nthCIS (n - 1) (tail())
zipWithCIS f cisa cisb =
  let loop (CIS a taila) (CIS b tailb) =
        CIS (f a b) <| \ () -> loop (taila()) (tailb())
  in loop cisa cisb
```

```
nthFib n =
  let fibs() = CIS 0 <| \ () -> CIS 1 <| \ () ->
                 zipWithCIS (+) (fibs()) (tailCIS (fibs()))
  in nthCIS n (fibs())
```

If there were a library/package implementing CIS's, this code is almost as short as the equivalent in Haskell above; however, for both they are likely slower than the first recursive implementation because of all the nested calls to "thunks" (a function that is a "closure" as in capturing external value bindings that is called with only the `()` argument - only a nothing argument). It wouldn't be that much slower as there are only three function calls per iteration if the "CIS" were memoized as lazy lists can be in Haskell, but Elm doesn't have deferred memoization so it runs into the same factorial performance problem as the non-TCP version. So while this can be done in Haskell, don't do it in Elm unless the language has lazy memoization added back into it. In other words, without memoization, this implementation is just a wordy version of the BAD non-TCP recursive version. Even in Haskell, this version will usually be slower than directly using recursion although not factorial performance, and Haskell may optimize away some of the "thunk" function calls.

What can One See from the Examples

One thing that should be clear in the above examples is that there are no statements in a "pure" functional language but only expressions, in that everything takes one or more inputs to produce an output or is a value which is basically a constant as it can never change. Thus, an `if..then..else..` in Elm is an expression that takes a condition and produces one of two outcomes based on the condition being true or false with both outcomes having to have the same type, as in if an integer is produced by one branch, one can't have a string or a float produced by the other. This is true also for the other `case ... of ...` branching expression which can be considered a multi-way branch.

Another thing that is clear is that arguments are separated from their function and each other only by white space (no round brackets and comma separators) for reasons to be explained later; thus, the `()` is just another argument that represents a nothing argument.

Finally, a lot of advanced things have been introduced in the last Haskell-like example in that a new Custom Type (CIS) has been defined with a type signature with a recursive definition that says it consists of the generic type `a` and a "thunk" function (the `->` indicates a function) that returns an instance of itself from an argument of `()`; it uses pattern matching where the contents of the type are extracted by using the "pattern" of its Constructor, CIS (both the type name and the Constructor named "CIS", but could be different Name's); anonymous functions/"lambda"'s (defined with the `\` symbol) were defined and used; also, the example uses forward (`|>`) and backward (`<|`) function "pipes" where the result of the last function is passed/piped to the last argument of the next function in the indicated processing flow direction.

The Details of Elm Syntax

Much of the Elm syntax is similar to many other languages: integer and float literals, UTF-8 character literals designated by `'a'` (representing the first lower-case letter of the English alphabet), `Bool` literals with the values of `True` and `False` (note the capitalization because they are Constructors), string literals by `"..."` or `"""..."""` with the latter form so as to contain unescaped `"`'s as well as unescaped new lines and line feed characters, the common mathematic symbols with a few differences as can be seen in the reference manual. Here the things that are different for functional languages as compared to most imperative

languages as in type definitions and aliases as well as type annotations/signatures, patterns and the use thereof in pattern matching, and also function currying and partial application will be covered.

Elm has some built-in types that are similar to as found in some other languages: the first is the tuple type (which isn't named but known by two or three items inside `( ..., ... )` round brackets separated by commas and can have different types contained therein), record types defined by `{ ..., }` curly brackets which have any number from zero up of named fields, each of its own type separated by commas, and the `List` type which contains zero of more elements of the same type. Examples of each definition with a type alias to make it easy to refer to is as follows:

```
type alias MyTuple = (Int, String)
type alias MyRecord = { int : Int, str : String }
type alias MyList = List Float
```

These can be nested to any depth and can contain each of any other types to any complexity. List's in Elm are actually `List a` where the `a` type variable stands for any type (thus generic) where it is here designated that the element type must be `Float`. One extension to the Record type is that when there is a type alias that immediately references a Record type as `MyRecord` above, one can use Constructor syntax to create an instance of that Record as in `MyRecord 42 "data"` where the Constructor arguments are in the same order as the Record definition rather than `{ int = 42, str = "data" }` (with the field assignments in any order). Record types also have some extensions as in update syntax where the new copy of a record only needs to have a few fields changed from the old version and extensible Record syntax where pattern matching is desired for any Record that contains **at least** the given fields as in the following code:

```
type alias MyRecord = { int : Int, str : String }
rec = MyRecord 42 "data" -- use Constructor
newrec = { rec | x = 7 } -- same as rec except update the x field
-- matches on any Record such as MyRecord that has an `x` field:
doit : { r | x : a } -> a -- a Record type `r` with a `x` field
doit rec = rec.x -- `rec` represents the matched any record
-- .x : { r | x : a } -> a -- "point" record accessors are defined this way
```

Note that use of "extensible record syntax" as in the last lines above results in a "generic" function unless it is limited via a type signature/annotation; this is one of the "gotchas" in the use of Records in that, unless one is careful, one may not get the record of the type they thought!

The only other general type in Elm is the Custom Type as in the following example:

```
type Custom a = Full Int a | Empty
type Tuple4 a b c d = Tuple4 a b c d
type NewList a = ConsNewList a (NewList a) | EmptyNewList
```

which last is a new list type with a generic element `a` that can contain an (possibly key or index) integer and one of the `a` types **or** the Empty variant that contains nothing at all. These are like "unions" in imperative

languages (actually tagged unions), and each "variant"/tag may contain anything including other types or its own type with as many types contained as required. One can build Tuple's of more than three items using a Custom Type as per `Tuple4` in the above example, and if one wants named fields, one can just wrap a record inside a Custom Type. Thus, it can be considered that Tuple's, Record's, and List's, are just special built-in cases of custom types.

Functions, values, and type definitions, can have type signatures or annotations as in the definition for the Custom Type `CIS` in that example where the second field is a function with the type signature `() -> CIS a`, meaning it is a function that takes a nothing/`()` and returns another `CIS a`, with the `->` separating the arguments and the result. An add function can be defined with its type signature as follows:

```
add : number -> number -> number
add x y = x + y
```

which takes two numbers and returns a number where `number` stands for either an Int or a Float but must be one or the other through the function application. Elm has several of these "type limiters"/contexts, where `appendable` means either List or String, `comparable` means just about anything other than functions, records, and custom types (tuples and lists work if the enclosed types are comparable), and `compappend` which means lists and strings. When it is desired to use more than one type using this "contraints"/type limiters, one can append a number to them, so `dummy : appendable1 ->appendable2 -> (appendable1, appendable2)` takes one kind of `appendable`, either a String or a List and optionally another kind of `appendable` to make a tuple of both kinds, which may be different.

While record fields can be accessed with their field names just as imperative language `struct`'s, the field accessors can also be used as "point" functions so rather than `mine.f`, one can use `.f mine`. However, pattern matching can be used with all of the tuple, record, and custom types as shown following:

```
type alias Mine = { f : Int, s = String }
type Custom a = Full Int a | Empty
test ((x, _) as arg0) ({ f, s } as arg1) ct =
  case ct of Full i c as fll -> (i, x)
             Empty as e -> (0, f)
```

This shows that for a custom type with variants, total coverage must be provided by a `case...of` expression, records can be pattern matched only on the exact spellings of all of their fields, tuples can be pattern matched as expected, and all pattern matches may optionally have the whole pattern named when designated by the `as` keyword. Any of the matched fields except for those of records can have their fields replaced by the `_` character meaning "ignore"/"unused" and as well, an entire pattern may be replace by this character if the entire pattern is to be ignored as in case targets or for nested patterns since pattern matches can be nested indefinitely.

Finally, functions in functional languages like Elm are different that as in imperative languages in that one doesn't have to pass the full set of arguments in order to call a function with the result being a function with the remaining arguments. Calling a function with less than the full number of arguments is called "partial

application" and generally calling functions without an enclosing designation is called "currying". For example, the following code:

```
add : number -> number -> number
add x y = x + y
add1 : number -> number
add1 = add 1
-- add1 v = (add 1) v -- same thing but redundant `v`
```

uses partial application of only `1` to `add` to produce a new function called `add1` which takes a number and produces a number which is one higher, but now the add1 function can be used without calling with the `1` at all!

Constructors for Custom Types and records (if there is a type alias for the particular record type) can also be partially applied just as regular functions can.

Advanced Concepts

As imperative programmers, there is no need to be too aware of "Category Theory" concepts even though Elm has them, just not under the usual names as in "Functor", "Applicative", and "Monad", and types could be extended past these. All of these deal with handling data that is wrapped in some sort of container such as the `Maybe` Type which is a tagged `union`/"variant" type with two variants: either a `Just` variant which contains some type of data or a `Nothing` variant that contains what it says. So that data type is wrapped inside a `Maybe` container called `Maybe`. Functional languages usually provide a `map` function that extracts the data type from the container (if it exists), applies a function to it, and returns the result wrapped in the same type of container; this is exactly what `fmap` does for Haskell's "Functor"'s. They may also provide a `map2` function that unwraps the contents (if they exist) from two containers, apply a function that takes two arguments to them, and wraps the result of the function in a new container of the same type; this is the same as `liftA2` for Haskell's "Applicative"'s. Finally, an `andThen` function may be provided that unwraps the contents of the container (if it exists) and apply's a function to it whose result must be already wrapped in the same container type; this is exactly like the `bind` function in Haskell's "Monad"'s. The difference, other than the function names, is that Elm doesn't have type classes (a way of implementing function overloading as per imperative languages), so each new Type must be inside its own module and define its own same-named functions for any of these it intends to support. The Haskell `return` function that wraps a value in its Monad container must be specialized in Elm without type classes, so is named as applied to the type as in the `Just` constructor for `Maybe`, the `Ok` Constructor for `Result`, and the `succeed` function for `Task` where the Constructors are private. These functions are all about the containers, just as those Haskell functions are!

Note that the Haskell `apply` function for "Applicative"'s can be implemented as `liftA2 id` or `map2 identity` in Elm with type signature as `Container (a -> b) -> Container a -> Container b` or in words, it is just like `fmap`/`map` except that the function is also wrapped and needs to be unwrapped before application. This completes all the functions necessary to do anything with containers that can be done in Haskell, although more verbosely because of the lack of function overloading/type classes.

So now we know how to create and perhaps use them, what about how do we control "side effects" which is the main property of "pure" functional languages? Haskell has an `IO` container for this, and Elm has the kind

of equivalent `Task` container with the Elm extension that it doesn't have exceptions so failure is described by the `Task` container. In other words, Haskell's container type is `IO a` with errors causing exceptions, whereas Elm's container type is `Task x a` where `x` represents the failure/problem type returned on error just as it does in `Result x a` which is another "monadic" container that can express the result of functions that might fail, so `Task` has the same failure/problem reporting mechanism as `Result` does. Another difference is that in Haskell there is only (normally) one chain of `IO` actions that starts with the `main` and runs to the end of the chain as started and branched to return to `main`, whereas in Elm there can be several chains of `Task`'s that get started in a `main` program or in its branches which all start with the program initialization code or in the various branches of the `update` function(s) where each branch can start a new `Task` chain that leads through a `perform` (never fails) or `attempt` (may fail) to a `Cmd` that is passed to the runtime code to execute the side effect and return the result to be incorporated into a state persisting "model" type (Elm's TEA - read up about it in the Elm Guide). Although most Elm code doesn't require this because TEA allows many `Task` chains to be run one after the other, it is possible to make a continuous chain of `andThen`'s and `map`'s that perform all the side effects of the program. For instance, the following code uses the "time" side effect module to time some pseudo expensive computation:

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

expensive : () -> Int
expensive() = List.range 1 10000000 |> List.length

testit : () -> (String, Cmd String)
testit() =
  ( ""
  , now |> Task.map posixToMillis |> Task.andThen(\ strt ->
      let rslt = expensive()
      in now |> Task.map posixToMillis |> Task.map (\stop ->
          "Result is " ++ String.fromInt rslt ++ " in "
            ++ String.fromInt (stop - strt) ++ " milliseconds."
    ) ) |> Task.perform identity
  )

main : Platform.Program () String String -- Model and Msg are String's
main = Browser.element { init = testit
                       , view = \ model -> text model
                       , update = \ str _ -> (str, Cmd.none)
                       , subscriptions = \ _ -> Sub.none
                       }
```

Some explanation is likely in order here: starting from the bottom, there are no subscriptions to externally triggered events so the subscriptions function is a do nothing, the update function just transfers the message string to the model string for display, the view function just puts the model string into a HTML text node, so all the work is done by the `testit` function started at initialization. Since the "flags" type is a nothing type, the "init" function takes that as argument then runs a `Task` chain starting with getting the time, converting that to an integer milliseconds value, then running the time expensive code, getting the

result and getting a new milliseconds time value, finally composing the message string from the result and the difference in time values which is passed to the `Task.perform` function to pass to the runtime as a `Cmd String`. The runtime gets the command, performs the task chain, and passes the resulting string as a message to the update function, which as above returns it as a model that the runtime then passes to the view function for display as HTML.

Although wordier than Haskell due to the lack of function overloading/type classes and no "do notation" syntax sugar, this is exactly how one would write this in the `main` of Haskell; in other words the `testit` function is pretty much exactly what the Haskell main would do written without using "do notation", with the rest unique to Elm's TEA system.

So how did this protect us from "side effects"? The `andThen`/`map` chain represents an unbroken chain of "actions" that are only described by the `Task` containers but are not run in user presented code but rather by the runtime. As a result, there are no "side effects" as far as user code is concerned, while preserving the requirements of functions always returning something: they return `Task`'s as per Haskell's `IO`, or pushing it a little further as here, they return `Cmd`'s to be handled by the runtime. Referential transparency as to the functions is also preserved because given a set of inputs, the code returns exactly the same output even though the runtime "side effects" may be different - the times obtained here are different for every run and the difference may be different so are definitely "side effects".

Conclusion

This article has covered the main differences and advantages in using Elm as compared to common imperative programming languages with examples showing those differences, but hasn't spent much time on language syntax or use of the language instructions. For more general coverage of the language syntax, the reader should refer to the Elm Language Reference document in this folder. For instructions about the use of the compiler and language, the reader should refer to the Elm Guide and documentation on the compiler repository GitHub site as well as the documentation for the various available libraries/packages.