Comparative Analysis of Functional Programming Languages

Martha María Nieto Ramírez

November 27th, 2023

Description

Objective: The goal of this task is to explore and compare two chosen programming languages—one purely functional and the other non-purely functional. The focus is on understanding key aspects of functional programming and the characteristics of each language.

"Pure" or "Mostly pure" language	“Impure” language
Lean	**Gleam**

History

Lean:

The Lean project was launched by Leonardo de Moura when he was at Microsoft Research in 2013. It is an open source project hosted on GitHub. Its mission is to tackle the challenges of scalability, usability, and proof automation in the Lean theorem prover.

Gleam:

Gleam was developed by Louis Pilfold in 2019. Gleam compiles to Erlang (or JavaScript) and has straightforward interop with other BEAM languages such as Erlang, Elixir, and LFE. The word “gleam” is a synonym of and rhymes with the word “beam”, and it’s easy to spell and say, so Louis thought it was a good choice. Gleam is an open source that prioritizes developer joy and productivity by leveraging Erlang's performance, reliability, and fault tolerance. Its strong interoperability with Elixir and Erlang fosters a vast code ecosystem utilization. Additionally, Gleam's robust type system and static analysis automate error detection, reducing manual debugging efforts.

Questions:

1. **Define a Named Function:**

Gleam	• Use the *pub fn* keyword to declare the function name and its parameters. The parameters are separated by commas and can have default values. The return type is specified after a colon. • Functions in Gleam are first class values and so can be assigned to variables, passed to functions, or anything else you might do with any other data type. • Write the function body using curly braces. The body can contain any valid Gleam code, such as variables, expressions, statements, or other functions. • Optionally, you can use type annotations to specify the types of the parameters and the return value. Type annotations are optional but recommended for clarity and documentation. `pub fn add(x: Int, y: Int) -> Int {` `  x + y` `}` **Can I create a named function outside a module?** Gleam does not support defining named functions outside a module. If we want to use functions from another languages, Gleam allow the importing of external functions.
Lean	• A function is defined using the def keyword followed by its name, a parameter list, return type and its body. • Each parameter is followed by a colon and its type. `def add1 (n : Nat) : Nat := n + 1` **Can I create a named function outside a module?** Yes, in Lean we can define a named function outside a module, but it becomes a global function.

2. **Define an Anonymous Function:**

Gleam	• We can define an anonymous function using the `fn` keyword. • Doesn't have a name and is often used when we need a function as a parameter to another function or for short, one-off functions. `// An anonymous function that takes two arguments and adds them` `let add = fn(x, y) -> x + y`
Lean	A lambda expression is an unnamed function. You define lambda expressions by using the fun keyword. A lambda expression resembles a function definition, except that instead of the := token, the => token is used to separate the argument list from the function body. As in a regular function definition, the argument types can be inferred or specified explicitly, and the return type of the lambda expression is inferred from the type of the last expression in the body. fun x => x + 1

3. **Define a Module:**

Gleam	Modules are defined by files and their directory paths. Each file is considered one Gleam module. `// Module` `src/directory/path.gleam` `// Importing in other module to use functions` `import directory/path`
Lean	In Lean, a module is essentially a Lean file. Each Lean file is its own module, and the module name is determined by the file name. The contents of the file define what is in the module. You can import other modules into a Lean file with the import keyword. `// Module` `src/directory/path.lean` `// Importing in other module to use functions` `import Mathlib.Logic.Basic`

4. Define a Type:

Gleam	• Custom types in Gleam are named collections of keys and values. Similar to objects in object-oriented languages, though they don't have methods. • Can be defined with multiple constructors, making them a way of modeling data that can be one of a few different variants. • Custom type variants can be pattern matched on using case expressions. • Custom types can also be destructured with a let binding. • Custom types can be parameterized with other types, making their contents variable.
Lean	In Lean, types are a first-class part of the language - they are expressions like any other. This means that definitions can refer to types just as well as they can refer to other values.

5. Write a Recursive Function:

Gleam	Recursion is most commonly found in recursive functions, which are functions that call themselves. A recursive function needs to have at least one base case and at least one recursive case. A base case returns a value without calling the function again, while a recursive case calls the function again, modifying the input so that it will at some point match the base case. Recursion is also used to loop in Gleam, as the language has no special syntax for looping. Instead, all looping is done with recursion. Gleam also supports recursive custom types, which have one or more of their variants refer to themselves in their contained data. This allows for more complex data structures to be defined in Gleam. `pub fn factorial(x: Int) -> Int {` `  case x {` `  // Base case` `  1 -> 1` `  // Recursive case` `  _ -> x * factorial(x - 1)` `  }` `}`
Lean	Using pattern matching Lean used the recursion mechanism to hangle cases. `def even (n : Nat) : Bool :=` `  match n with` `  | Nat.zero => true` `  | Nat.succ k => not (even k)`

6. Lazy or Strict Evaluation:

Gleam	Strict / Eager evaluation is the default strategy used my most programming languages. In eager evaluation, an expression is evaluated as soon as it is bound to a variable: let x = 1 + 2 let y = x * 3
Lean	Lean employs lazy evaluation as its evaluation strategy. It is a purely functional language that uses lazy evaluation. This means that expressions are only evaluated when they are needed, and not before. This allows for more efficient computation and avoids unnecessary work.

7. Static or Dynamic Typing:

Gleam	It's statically typed. This means that the type of a variable is known at compile time. The main advantage here is that all kinds of checking can be done by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.
Lean	Lean is a statically typed programming language. This means that every expression in Lean has a type that is known at compile time, and the compiler checks that the types are consistent. Static typing helps to catch errors early, and also enables various optimizations and code analysis tools. Lean's type system is based on the calculus of constructions with inductive types, which is a powerful logical framework that allows for the definition and manipulation of complex data structures and proofs. Lean's type system also supports dependent types, which are types that can depend on values.

8. Project Creation Commands:

Gleam	1. `gleam new project_name` 2. `cd project_name` 3. `gleam run` 4. `gleam test` 5. `gleam shell` // Alternative to run an Erlang shell.
Lean	`$ lake new hello` `$ cd hello`

9. Computational Notebook Usage:

Gleam	Apparently, there is no explicit mention of Gleam supporting computational notebook environments like Jupyter. However, Gleam can compile to JavaScript, enabling you to use our code in the browser or anywhere else JavaScript can run.
Lean	There is no mention of supporting computational notebook environments.

10. Browser-Based Execution:

Gleam	There is no online platform to try Gleam without installation.
Lean	Yes, there is an online version providing recent versions of Lean 4, std4 and mathlib4.

11. Concurrency Support:

Gleam	Gleam provides concurrency. It runs on the Erlang virtual machine (BEAM), which is known for its highly concurrent, fault-tolerant capabilities. Gleam takes full advantage of the Erlang runtime. All Gleam programs are as fast and as efficiently multi-threaded as their Erlang counterpart. Gleam supports a multi-core actor-based concurrency system that can run millions of concurrent tasks. This, along with fast immutable data structures and a concurrent garbage collector that never stops the world, allows your service to scale and stay lightning fast with ease.
Lean	Lean is a purely functional programming language, and it does not provide built-in concurrency support.

12. Pattern Matching:

Gleam	Pattern matching is a powerful feature in Gleam that allows us to destructure and match data. In Gleam, let = can be used for pattern matching. The case expression is the most common kind of flow control in Gleam code. It allows us to say "if the data has this shape, then do that", which is call pattern matching. `case some_number {` `  0 -> "Zero"`

	`` 1 -> "One" 2 -> "Two" n -> "Some other number" // This matches anything } ```
Lean	In Lean, pattern matching is implemented using the `match` keyword. The `match` keyword takes a value of a specific type as its argument and returns a list of pairs, where each pair consists of a pattern and a value of the same type. The pattern is a Lean term that is used to match against the value, and the value is the value that the pattern is matched against. ``` def even (n : Nat) : Bool := match n with | Nat.zero => true | Nat.succ k => not (even k) ```

13. Error Handling:

Gleam	For error handling is typically done using the Result type. The Result type is a way of representing computations that can fail. It has two variants: Ok for successful and Error for computations that have failed. The case expression is then used to pattern match on the Result. If it's an Ok, it prints the parsed integer. If it's an Error, it prints an error message. There are also libraries for working with errors and computations that can fail, such as gleam-experiments/snag, which provides a boilerplate-free ad-hoc error type. ``` case parse_int("123") { Error(e) -> io.println("That wasn't an Int") Ok(i) -> io.println("We parsed the Int") } ```
Lean	• Using Option and Except types: These types represent programs that can fail by returning none or an error value. They can be used with andThen and ok helpers to chain and handle potential failures. • Using WithLog type: This type represents programs that accumulate a log while running. It can be used with andThen, ok, and save helpers to record and pass along extra information. • Using State type: This type represents programs with a single mutable variable. It can be used with andThen, ok, get, and set helpers to simulate local mutable state.

	`` inductive Except (ε : Type) (α : Type) where | error : ε → Except ε α | ok : α → Except ε α deriving BEq, Hashable, Repr infixl:55 " ~~> " => andThen def firstThird (xs : List α) : Except String (α × α) := get xs 0 ~~> fun first => get xs 2 ~~> fun third => ok (first, third) ```

14. Community and Ecosystem:

Gleam	Its community is growing but its not as large as some of the more established languages. • GitHub: 352 followers. • Twitter (X): 2,078 followers. • Discord In terms of libraries and frameworks, Gleam makes it easy to use code written in other BEAM languages such as Erlang and Elixir. This means there's a rich ecosystem of thousands of open source libraries for Gleam users to make use of. The Gleam Package Index and the Awesome Gleam resource list are good places to find Gleam libraries. The Awesome Gleam GitHub repository is a collection of Gleam libraries, projects, and resources.
Lean	The Lean programming language has a diverse and active community that primarily gathers on Zulip chat and GitHub1. However, the exact size of the community is not readily available. • GitHub: 299 followers. • Twitter (X): 3,321 followers. • Mastodon: 276 followers. In terms of libraries and frameworks, Lean has a community-maintained project called mathlib that began in 2017. The goal of mathlib is to digitize as much of pure mathematics as possible in one large cohesive library, up to research-level mathematics. As of November 2023, mathlib had formalized over 127,000 theorems and 70,000 definitions in Lean.

15. Interoperability:

Gleam	Gleam can easily interoperate with other BEAM languages such as Erlang and Elixir. This means that Gleam code can be used by programmers of other BEAM languages, either by transparently making use of libraries written in Gleam, or by adding Gleam modules to their existing project with minimal fuss. In addition, Gleam can compile to JavaScript, enabling you to use your Gleam code in the browser, or anywhere else JavaScript can run. This allows for some level of interoperability with JavaScript and TypeScript. Gleam has Cheatsheets for using Gleam in Elm, PHP, Python, Rust and of course Elixir and Erlang.
Lean	• Client-extension and Language Server Protocol server: Lean interfaces with other systems via a client-extension and Language Server Protocol server. This allows it to interact with different programming environments. • Compilation to JavaScript: Lean can be compiled to JavaScript and accessed in a web browser. This allows it to be used in web development contexts alongside other languages like HTML and CSS. • Extensive support for meta-programming: Lean's support for meta-programming can also aid in interoperability.

References:

GLEAM:

https://gleam.run/book/tour/index.html

https://learn.codesee.io/interview-with-louis-of-gleam-a-language-for-building-type-safe-scalable-systems/

https://gleam.run/

https://github.com/gleam-lang/gleam

https://github.com/gleam-lang

https://exercism.org/tracks/gleam/concepts

https://github.com/gleam-lang/awesome-gleam#error-handling

https://hexdocs.pm/gleam_otp/

https://gleam.run/documentation/#cheatsheets

LEAN:

https://lean-lang.org/about/

https://lean-lang.org/lean4/doc/functions.html

https://www.youtube.com/watch?v=S-aGjgIDQZY

https://lean-lang.org/functional_programming_in_lean/title.html

https://lean-lang.org/introduction_to_lean/introduction_to_lean.pdf

https://github.com/leanprover/lean4/tree/master/src/lake

https://live.lean-lang.org/

https://lean-lang.org/functional_programming_in_lean/monads.html#propagating-error-messages

https://leanprover-community.github.io/