**Lua Feasibility Research**

**Lua Academic Paper**

* *“Extension languages should have a clear and simple syntax, because their main users are not professional programmers.”* – Research into Roblox scripts should show how well this design philosophy works*.*
* *“Extension languages are not for writing large pieces of software, with hundreds of thousands lines. Therefore, mechanisms for supporting programming-in-the large, like static type checking, information hiding, and exception handling, are not essential.”* – Game mods can be quite large, so perhaps some of these features would be convenient when writing large mods (e.g. DBM).
* By design, Lua is a **general purpose** extension language, and is therefore not in any way designed for the development of game modifications (mods).
* Lua has several meta mechanisms which allow it to be so extensible:
  + **Dynamic associative arrays:** Implement a multitude of data types, including ordinary arrays, records and sets. Lever the data description power of the language by means of constructors.
  + **Reflexive facilities:** Allow for the creation of highly polymorphic parts. For instance, whilst persistence and multiple name spaces are not present in Lua, they can be implemented via reflexive facilities.
  + **Fallbacks:** Used to extend the meaning of syntactical constructions, e.g. can be used to implement different kinds of inheritance.
* *“Lua has no notion of a* ***main*** *program”* – This makes sense for game mods so the proposed new language should probably maintain this concept.
* *“Lua is provided as a library of C functions to be linked to host applications.”* – If Lua is already interpreted as a library of C functions, then it should be fairly simple to translate the proposed new language to C too.
* Lua has a **global** scope, which I may want to carry forward to the proposed new language.
* *“The unit of execution of Lua is called a* ***chunk****. A chunk may contain statements and function definitions. When a chunk is executed, first all its functions and statements are compiled, and the functions added to the global environment; then the statements are executed in sequential order.”* – Due to this execution model, Lua in a sense is Event*-*Driven by the host application. I think this model of execution likewise makes sense for mods.
* *“Lua is a dynamically typed language.”* – As a scripting language, this seems like the correct choice. For larger mods however, perhaps it does not make sense to use a dynamically typed language, so this is something that should at least be considered.
* *“Functions can return multiple values, and multiple assignments can be used to collect these values.”* – This is really a language design choice, but perhaps something worth changing if we can think of a good reason to.
* *“Functions in Lua are first class values.”* –This is a very good language feature that I think should definitely be maintained.
* Types:
  + number (floating point values)
  + string
  + function
  + nil – essentially null, and is the default value of an uninitialized variable.
  + userdata – arbitrary host data, the only valid operations on values of this type are assignment and the equality test.
  + table – Associative arrays, i.e. arrays that can be indexed not only with integers, but also with strings, reals, tables, and function values.
* **Associative Arrays:**
  + Constructor functions exist in either C or Lua that are passed a table object and then perform the appropriate operations on this newly created object.
* **Reflexive Facilities:**
  + Tables are the intended way of maintaining multiple environments.
  + Syntactic sugar exists that allows for support elements of OOP.
* **Fallbacks:**
  + *“Being an untyped language, Lua has a semantics with many run-time abnormal conditions. … Because halting in these situations would be unsuitable for an embedded language, Lua allows programmers to set their own functions to handle error conditions; such functions are called fallback functions.”* – It is a key point to acknowledge that any errors with the proposed new language must be handled without causing the host application to crash.
  + Types of fallback:
    - **"arith", "order", "concat"** – When an operation is applied to invalid operands.
    - **“index”** – When Lua tries to retrieve the value of an index that doesn’t exist, by default this simply returns nil.
    - **"gettable", "settable"** – When Lua tries to read or write the value of an index to a non-table value.
    - **"function"** – Called when Lua tries to call a non-function value as a function.
    - **“gc”** – Called during garbage collection, and by default is unhandled.
  + *“Fallbacks are used mainly by expert programmers when binding Lua to a specific application. After that, the facility is used as an integral part of the language.”* – This probably means that most games have fallbacks already implemented by the developers themselves, and as such should be considered greatly when designing the proposed alternative language solution.
  + An inheritance model for Lua can be implemented via fallbacks, which allows for an immense amount of variety in how it is implemented – definitely something to keep in the proposed new language.
* Other extension languages:
  + **Lisp** (especially **Scheme**)
  + **Tcl**
  + **Python**

**Addon Code Analysis**

* The **local** keyword is repeated a lot. I really think considering how modular the bigger mods are, it makes sense for **global** to have to be explicitly defined rather than the other way around.
* The **end** keyword also repeats a lot to end function and block (if statement) scope.
* For some larger addons, the core **.lua** files are massive and way beyond the scope that Lua was initially designed for. However, it is still possible to split code up into appropriate modules.
* I find the colon in *object name* ***:*** *function* to be a pretty strange syntax. This is needed if you don’t want to use the **self** keyword in the function scope to access other attributes of the object in question.
* **function(…)** indicates that the function can take a variable number of arguments. These are put into a table called **arg**.
* It might make sense to base the syntax mostly on Lua, but introduce more syntax sugar and more features, possibly more functional concepts such as streams.
* It also might make sense to determine the crucial concepts of the domain as these should be deeply incorporated into the design of the DSL.
* Side note: So many of these Roblox scripts (*exploits*) are obfuscated to obscure malicious behavior it’s unreal.
* Now I really am being picky, but multi-line Lua comments are created by **–[[** and **]]—**to close it, and it’s just so unnecessary.
* Lua doesn’t have normal arrays, just associative tables. This means that iterating over a list as you would in array requires something like: for \_, value in pairs(array) do or for \_, value in ipairs(array) do (if you want to maintain the index order), which is not great in my view.
* Small things, but I’d rather have **null** than nil**.**

**To-do**

1. Categorise the mods by what they are doing for the game and check for similarities.
2. Look at what the conditionals are actually checking for.
3. Ask mod making communities/specific mod makers for their thoughts on Lua, and how they would design the hypothetical DSL.