|  |  |
| --- | --- |
| **Andrew Boorde**  **Justin Le**  **Ross** [**M**](mailto:aboorde@uncc.edu)**cGuinn**  **Jonathan Suttles** | [**aboorde@uncc.edu**](mailto:aboorde@uncc.edu)  [**jle5@uncc.edu**](mailto:jle5@uncc.edu)  [**mcguinn12r@unccc.edu**](mailto:mcguinn12r@unccc.edu)  [**jsuttle4@uncc.edu**](mailto:jsuttle4@uncc.edu) |

**ITCS - 4102 Programming Languages**

**Lua Term Project**

**Fall 2014**

**Language Paradigm**

Although it is primarily a procedural language, Lua has the ability to be used for a variety of paradigms depending on how the user wishes to implement the language, as Lua does not force the user to use a specific object or class model. This allows for any paradigm style to be implemented with no modifications needing to be done to Lua. For example, an object-oriented paradigm can easily be achieved through encapsulation of the table data structure in Lua, allowing for the addition of objects with no changes to the language itself.

Lua can even support a functional paradigm even though the basis of the language is procedural. This is due to the lexical scoping and first-class functions, which allow functions to be treated like any other value, necessary for a functional paradigm. The flexible nature of Lua gives the language a powerful set of features which adapt to the programmer’s specification.

**Lua History**

Lua was created by a small committee of three members from Brazil in 1993. Extremely harsh trade restrictions imposed by the Brazilian government made it nearly impossible to afford hardware and software created outside of the country. Tecgraf, a Brazilian company specializing in graphics technology and known for the creation of Lua, originally created two separate languages for their applications, DEL and SOL, which became the groundwork and inspiration for Lua.

DEL and SOL were the languages used for an application created for a Brazilian oil company in order to make programming their necessary simulation input more efficient and reliable. DEL, which stood for data-entry language, was a domain-specific declarative language created to define task specifications, such as input and output. What made DEL so useful was that the language also created a graphics file containing the diagram of specifications. SOL, meaning Simple Object Language, was a specialized declaration language used to create type declarations and objects.

In a meeting to discuss the future of the two languages, the teams that created them realized they can be combined to make a more powerful language. From this, Lua was designed to be a full language while maintaining a small, portable implementation. Over the past 20 years, Lua has seen many additions and improvements that have made it a widely accepted language. The second version of Lua brought extensible semantics which allowed for tables to be the basis for objects. Later improvements also introduced functional programming and additional control abstractions. Backed by a supportive community, Lua continues to evolve as a language and increase its impact and popularity.

**Language Elements and Syntax**

Lua has eight types: table, boolean, nil, number, function, thread, string, and userdata. Being a dynamically typed language, types define values, not variables. Variables, by default, point to a nil type until it is assigned another value to make it true, even 0 and empty strings.

In Lua, identifiers are used to name any user defined item and may start with and contain digits, characters, and underscores. The language is also case-sensitive, meaning even though two identifiers spelled the same Lua contains reserved words which are not able to be used as identifiers.

Here is a list of reserved words in Lua:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| do | while | and | if | elseif | else |
| then | nil | end | or | true | until |
| return | false | not | break | for | function |
| in | local | repeat |  |  |  |

Tables are the only structure type that exist in Lua, but don’t let that fool you. Any structure that you may find in another language, such as arrays, queues, lists, etc., are easily represented with tables in Lua. Tables are associative arrays that hold key/value pairs, where the key can be any value other than nil. Creating a table only requires a constructor expression, { }, with a variable holding no relationship to the table itself, only referencing to it.

**Basic Control Abstractions**

*If*

This is the only conditional control structure existing in Lua, unlike the looping abstraction that is present in the other control structures. The basic use of this structure consists of the keywords ‘if then end’. The program will pause when this structure appears in the code, checking to see if the ‘if’ condition is true or false. If true, the program executes the code that follows ‘then’ until the program arrives at ‘end’.

The functionality of this abstraction is expanded by including ‘else’ and ‘elseif’ into the control structure. When utilizing ‘else’, if an ‘if’ condition does not evaluate to true, the program will instead execute the code which follows ‘else’ until ‘end’ is given in the code.

The ‘elseif’ keyword is a condition that is only evaluated in the event the condition before it does not occur. Abstractions can be written to include multiple ‘elseif’ keywords which will be evaluated in order until a condition is met, executing only one ‘elseif’ condition.

*While*

The while abstraction exists as an iterative control structure, which has the ability to continue looping as long as the condition is met. When a program reads the ‘while’ keyword, it evaluates the condition that follows to determine its next step. If false, the program will move on to the ‘end’ keyword at the tail of the abstraction and continue from there. If the condition evaluates true, the program will execute what appears after the ‘do’ keyword until it sees ‘end’ or ‘break’. If there is ever a ‘break’ that is encountered, the program will exit the loop and keep running. When it reaches ‘end’, the program will reevaluate the ‘while’ condition at the beginning of the loop and continue this process until the condition isn’t satisfied or a ‘break’ is reached.

*For*

In Lua, the *for* control structure is an iterative loop which continues until the count hits the limit set in the condition. The ‘for’ abstraction takes in an initial value, value limit, and if the user chooses to include it, what the value will increment by each loop. When the program evaluates the *for* condition, it is checking to see if the counter is less than or equal to the set limit on the value. If true, the code after the ‘do’ keyword is executed until it reaches ‘end’ where the counter is incremented by the given value. This loop will end once the counter exceeds the limit when the condition is evaluated or a ‘break’ is encountered at any point in the ‘do’ execution.

*Repeat*

The *repeat* control structure is very similar to a *while* abstraction in that the loop will repeat its body until the condition is true. The difference lies in the fact that a *repeat* structure will test at the end of the loop following the until keyword. A *while* control structure tests the condition at the beginning of the loop. Therefore, not requiring the block to execute. A *repeat* control structure forces the code to execute at least once before checking if the condition is met.

**Abstraction Handling**

Functions represent a significant portion of abstraction for statements and expressions in Lua. They have the ability to exist as a procedure or return a value. Functions used in Lua can either exist in Lua or any language used by a parent program. They are also first-class values, which means a function has the same permissions as any other value in Lua. This allows a function to be passed as an argument, be stored as a variable, or returned as a value of a different function. Functions also have lexical scoping, giving users the ability to incorporate the functional paradigm properties if necessary.

On its own, Lua does not contain objects or the concept of classes. However, due to the flexible nature of Lua, we are able to define our own classes and create objects using tables. To create the class, a table would just have to include data and function fields. With this table, objects are created with a constructor function. As everything is an object in Lua, this even allows for inheritance rules to be created if needed.

**Readability, Writability, and Reliability Evaluation**

*Readability*

Lua provides a simple easy to read syntax based in C. With only tables used as a structure, the language possesses great Orthogonality, essential to a readable language, as written by Sebesta. With just one structure we can create a model of any other data structure needed.

There is some readability lost in the way types are defined. The dynamic nature of typing gives values types at runtime, with variables only referencing position. This can cause some issue in regards to simplicity and reading the language.

*Writability*

Lua is a language with very few keywords. This can be considered a positive facet of the language because it allows beginners of the language to jump in easier than if there were a larger amount of keywords, because if there is a long list of reserved words to be memorized and cannot be used as identifiers. Another aspect to note is the way blocks of code end. And it is that simple; they end with “end”! All and all, the syntax is pretty similar to the language of C.

However, some of these benefits can also be considered as pitfalls. Finally, the last major pitfall of Lua’s writability is that there are very limited data types.

*Reliability*

Lua provides dynamic, run-time type checking on it’s built in operations, but does not do so to parameters and return values of function calls. This gives functions the ability to take in different value types, but can cause errors if the function is called on the wrong type, reducing reliability.

Normally, due to the nature of Lua, the application program using Lua performs error handling. However, if needed, Lua provides a protected call function (‘pcall’) to handle errors within the language. This function works in protected mode, calling the encapsulated function and catching any errors while the function is still running. In Lua, exceptions are thrown with ‘error’ and caught with ‘pcall’.

Aliasing also exists in Lua, increasing its reliability. This allows the user to not have to change variable names in functions when they are called more than once with different variable names or when the user wishes to use a function from a previous work.

**Strengths and Weaknesses**

There are many strengths and weaknesses in lua. Since Lua is very small, it can be built into a 182k executable interpreter. It is written in C and It can be run on any platform with an ANSI C compiler. This makes the Lua language very portable. Lua is a very fast scripting language that has a JIT compiler, or a Just-In-Time compiler, which is a component of the Java Runtime Environment that improves the performance of java applications at runtime. Those who are still not satisfied with performance can write the important parts of their code in C, and integrate the C code into their Lua code.

Lua is also very well documented, and is built around a friendly and enthusiastic community. There are books, reference manuals, short references, and many more documentations that exist for Lua. It has clean and simple syntax which can help beginners and allows lua to be more accessible to non-programmers. It has lexical scoping and a simple, yet very powerful debugging library. There are many more strengths in Lua, but there are also some weaknesses.

**Programs Overview**

*Marble Clock*

This problem involved simulating multiple trays of a marble clock. As time progresses, the marbles fall through the trays representing different times. The specifics of how the marbles fall suggests that some rows should be implemented using queues and others should be implemented using stacks. Lua doesn’t directly implement either of these, but the table data structure is robust enough to reproduce the effects of both. This let us go ahead and just use tables instead of implementing different data structures. Because of this, it was a relatively simple program to write.

*Three of a Crime*

Three of a Crime is a logic puzzle where you try to deduce the names of 3 criminals. You’re given 3 names and told how many of them are criminals and how many are innocent, but not which are which. This obviously involved lots of input and output. The hardest part about coding the problem was learning how to use the console I/O. Compared to other languages, this was pretty difficult to do, mainly due to the different formatting options you have for input.