A Two-Way Pushdown Automaton for the Lua Language

by

Kevin Stevens

A Thesis Presented in Partial Fulfillment
of the Requirements for Graduation From
Barrett, the Honors College

Approved May 2020 by the
Barrett Honors Thesis Committee:

Yan Shoshitaishvili, Director
Ruoyu Wang

ARIZONA STATE UNIVERSITY

May 2020

ABSTRACT

A two-way deterministic finite pushdown automaton ("2PDA") is developed for the Lua language. This 2PDA is evaluated against both a purpose-built Lua syntax test suite and the test suite used by the reference implementation of Lua, and fully passes both.

ACKNOWLEDGMENTS

The author would like to sincerely thank Dr. Yan Shoshitaishvili of Arizona State University for suggesting, overseeing, and directing this project, as well as to Dr. Ruoyu Wang and Faezeh Kalantari for significant input and guidance.

Additional thanks is extended to Shahabudeen Sajith for help with analysis of the parsability of Lua long brackets.

Finally, the author would like to thank Barrett, The Honors College at Arizona State University, for providing the opportunity to perform this kind of research.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

A common form of web security vulnerability is remote code injection. These attacks occur when specific unusual inputs cause software running on a server to execute them as code, in a manner not intended by the programmer. Tools that can help researchers easily identify and characterize these forms of vulnerabilities are very useful for ensuring that they are discovered and fixed early. One current deficiency in this area is that after a possible remote code injection vector has been identified, it is not always easy to deduce what type of language construct the server is expecting (a statement? An expression? String escape sequences?), or, more broadly, to even identify the language that the input is being misinterpreted as.

One proposed approach for automating the code-injection language identification process is through the application of graph traversal algorithms to parsing automata to identify the server's most probable parsing state through repeated queries to the server. A first step towards performing such research is to create automata that can parse real-world languages.

Chapter 2

BACKGROUND

There are a few challenges to creating a new parser for any existing programming language, mostly stemming from inadequate or incomplete documentation about the specifics of its syntax. The most important of these are elaborated here.

2.1 Syntax Reference

The syntax of Lua is described in "The Complete Syntax of Lua" [2] in EBNF notation. This grammar is, however, incomplete in several ways:

- Descriptions of whitespace and comments are omitted.
- The symbols "Name," "Numeral" and "LiteralString" are defined in prose elsewhere in the documentation, not in the EBNF itself.
  - The prose explanations are often not precise enough; for example, the description of "Numeral" does not explain that a numeral can begin with no digits preceding the decimal point.
- Details relating to precedence and other ambiguities are not clarified.
- The grammar does not mention that if the first line in a Lua file begins with the "#" character, the line is ignored.

None of these complications are insurmountable, but they must be kept in mind when designing any new parser for the language.

2.2 Implementations

Since the official Lua documentation leaves some of the syntax details unclear, existing source code must be consulted to resolve ambiguities. While there are a myriad of existing implementations of Lua [1], the two most prominent are the original reference implementation (with the same name as the language itself), and "LuaJIT," which uses just-in-time compilation to provide significant speed benefits [4]. Any new parser implementation should be informed by both the official language reference and the source code from an existing interpreter or compiler.

Chapter 3

DESIGN

A two-way deterministic finite pushdown automaton ("2PDA") was developed, which recognizes version 5.3 of the Lua programming language [2]. Despite the obvious interpretation of the name, the "two ways" this automaton can move are to move right and to stay put. It cannot move left to reread previous input characters.

The program is split into three primary components: the abstract 2PDA, the Lua 2PDA, and the test suite.

Syntax ambiguities and unclear edge cases were resolved by consulting the source code for the reference implementation of Lua. This was chosen over LuaJIT and other implementations not only merely because it is the reference implementation, but also because the code for LuaJIT is more complicated due to implementing a full just-in-time compiler instead of an interpreter.

3.1 Abstract 2PDA

The file "pda.py" contains a Python class representing an abstract 2PDA. This class does not include any specific transitions tying it to any specific language, and focuses instead on code for parsing strings, using transitions defined in a subclass.

Transitions are defined in a Python dictionary, with one key-value pair per transition. Each key in this dictionary is a 3-tuple: "(current state, character, value on top of stack)." The "value on top of stack" can be set to `None`; in this case, the transition will be taken only if there is no other transition that does match the current value on top of the stack. Dictionary values are 4-tuples: "(target state, movement action, stack action, new stack value).” A parse error occurs when there is no transition from the current state matching the current input string character and stack value.

There is no explicit list of available states, though one can be generated from the transitions dictionary. States are defined only by having a transition to them.

3.2 Lua 2PDA

The file “pda_lua.py” contains a subclass of the abstract 2PDA class, which adds states and transitions representing the Lua language. The class defines over 1400 states, 560,000 transitions, and 200 stack symbols.

As previously noted, parse errors are defined as the lack of a transition matching the current situation. For situations where a parse error is needed but some sort of transition is required to catch it, the Lua 2PDA uses a general-purpose “FAIL” transition, which points to a state of the same name and does not increment the input head. This state has no outgoing transitions, and thus is guaranteed to cause an immediate parsing failure right after it is taken.

3.2.1 Subsystems

The Lua 2PDA is organized into a set of distinct “subsystems,” representing different parts of the language syntax. These subsystems “call” each other (in a sense) recursively in order to parse nested syntax constructs.

The subsystems are described here roughly in order from most specific to least specific – that is, the later subsystems build upon the earlier ones.

3.2.1.1 Entry/Exit Functions

Some of the subsystems described below are simply accessed by transitioning to a specific state with a known name, but for others that are more complicated, helper functions that create various transitional states are used. These generally accept as input an entrypoint state to transition from, and one or more exit transitions to use when returning from the subsystem. Usually there is one primary exit, and several extra exits representing unusual cases that are not necessarily errors.

These functions generally keep track of the state to return to by pushing the name of the source state onto the stack when the subsystem is entered. This is used instead of the destination state name because multiple calls to an entry/exit function may point to the same destination state, and because the destination state does not contain all of the information about the return transition (such as what stack operation to perform) anyway. When the subsystem is exited, the value on top of the stack is matched, and the appropriate exit transition specified by the caller is taken.

Some of the subsystems return one or more values on top of the stack to convey some extra information to the caller about what was parsed. Since the 2PDA can only ever see the value on top of the stack, these get in the way of selecting the proper return transition. This is addressed by popping these extra stack symbols off of the stack and temporarily mangling them into state names. The number of additional states needed to perform this maneuver for a single function call increases by a factor of N for every additional stack symbol (where N is the number of possible values that that stack symbol may have); however, with the most complicated subsystems only returning a maximum of three stack values at a time, this never grows large enough to be problematic in practice. Once all of the information from the stack symbols has been transferred to the state, the value dictating the return transition is read, the stack values are recreated, and the return occurs.

3.2.1.2 Long Strings and Multiline Comments

This subsystem is used to parse constructs that use “long brackets;” specifically, “long-form strings” and “long comments.”

This subsystem does not have any input or output, and thus does not use an entry/exit function. There are two entrypoints, one in which the client code has already read the first “`[`”, and one where it has not.

Lua’s long-brackets feature is interesting in the context of PDA design because it actually cannot be properly modeled by a 2PDA. Long brackets are a pair of square brackets separated by some number of equals signs; for example, “`[===[ content goes here ]===]`”. Only closing long-brackets with a number of equals matching the opening set will match, which is convenient for nesting. This, however, cannot be recognized by a deterministic 2PDA, since matching a closing long-bracket requires popping equals signs off the stack, and if the number does not match, the stack cannot be rebuilt, because the 2PDA is unable to move back to reread the number of opening equals signs.

While it is not theoretically possible to correctly model long brackets with a 2PDA in the general case, it is possible to create enough states to properly handle varying numbers of equals, up to some arbitrary but finite limit chosen by the 2PDA designer. The implementation presented here correctly handles up to 10 equals signs, and the limit can be adjusted easily via a constant if desired. In addition, the reference implementation of Lua stores the number of read equals signs in a C variable of type `size_t`, meaning that there is a practical (albeit massive) limit to the number of equals signs on any given platform anyway.

Long strings do not interpret escape sequences; all backslashes are preserved. This makes them very easy to parse, apart from the issue with long brackets described above.

3.2.1.3 Whitespace and Comments

This subsystem parses Lua whitespace and comments. These are not included in the “Complete Syntax of Lua,” and are just implied to be legal between any tokens. Comments are permitted in Lua anywhere whitespace is.

This subsystem uses an entry/exit function that transitions back to the starting state once a non-whitespace-or-comment character is encountered. This is convenient because one can just call this function in between tokens, without needing to define extra states for before and after the whitespace. This system reads single-line comments directly, and “long form” comments through the help of the “multiline comments or long strings” subsystem (described above).

Single-line comments in Lua are denoted by two minuses (“`--`”). This can prove tricky to parse, because the whitespace system needs to fully consume a “`-`” if it encounters one, since it may be the start of a comment. However, whitespace is also allowed in expressions, and a “`-`” by itself in an expression represents the binary operator minus. The 2PDA cannot back up (only move forward or stay put); thus, if it reads a “`-`” followed by something else, it needs to be able to return to the calling code while conveying this information, since the “`-`” may be meaningful in context.

This is done through a second, “minus” exit. If a “`-`” is consumed but not followed by another “`-`”, the system will exit using that transition. In many calls to `read_whitespace()`, the “minus” exit is just set to the “FAIL” transition, because minus signs are not legal in most places. However, since most parts of the Lua grammar can be found (directly or indirectly) within expressions, and minus signs in expressions refer to the binary operator for subtraction, most other subsystems include their own “minus” exits that are transferred to their final calls to `read_whitespace()`.

3.2.1.4 Names and Keywords

This subsystem parses both names and keywords. “Names” refers to the “`Name`” symbol as used in “The Complete Syntax of Lua.” It does not read leading or trailing whitespace; that is left to the calling code.

A single system for reading both names and keywords is needed because it is not generally possible to know whether a string of alphanumeric characters represents a name or keyword until it has been fully read. For example, anything beginning with “`z`” can be immediately classified as a name (since no keywords start with that letter); however, “`e`” could be the start either of a name, or of the “`else`” or “`elseif`” keywords. Even if a partial reading produces “`else`”, the 2PDA still does not know which of the keywords it is going to be. Furthermore, even reading “`elseif`” does not guarantee that the token is a keyword, since, for example, “`elseifz`” is a hypothetical valid name. Since keywords are never valid as names, names and keywords are always handled differently in Lua; thus, having a single system that reads and classifies a sequence of alphanumeric characters is very useful.

This subsystem uses an entry/exit function with two exit transitions: one used if the token was a name, and one used if it was a keyword. In the latter case, a stack symbol for the specific keyword that was read is left on top of the stack.

The system works by assembling a keyword on the stack character by character (using existing stack symbols representing partially-read keywords) for as long as the input matches a known keyword. If a character does not match what would be expected for any Lua keyword, or another character is read following the last character in a known keyword, the 2PDA pops that stack symbol and transitions to a state that just continues reading until a non-alphanumeric character is encountered. In that case, the “name” exit is used. However, if a non-alphanumeric character is read at the end of a keyword, the "keyword" exit is used instead, with a symbol representing the keyword left on top of the stack.

3.2.1.5 Name Lists

This subsystem parses "name lists," corresponding to the "`namelist`" symbol in "The Complete Syntax of Lua."

Like the name-or-keyword subsystem, this system includes two exits, one for "name" and one for "keyword." The "name" exit is taken if one or more names (separated by commas) are read, and the "keyword" exit is taken if the first alphanumeric token read is actually a keyword. The latter exit is used by statements starting with the keyword "`local`", which can be followed by either a name list or the keyword "`function`."

This subsystem consumes all trailing whitespace. It must do so, because there can be an arbitrary amount of whitespace between names and commas, so it must read all whitespace following each name and comma to see if another comma or name (respectively) follows it or not. Many of the other subsystems also consume trailing whitespace for the same reason.

3.2.1.6 Function Bodies

This subsystem parses function bodies, corresponding to the "`funcbody`" symbol in "The Complete Syntax of Lua." A function body consists of a parameter list, and a block of statements ended by the "`end`" keyword. The subsystem has no input or output, and does not use an entry/exit function.

3.2.1.7 Short String Literals

This subsystem parses short string literals, corresponding to one production of the "`LiteralString`" symbol in "The Complete Syntax of Lua." Unlike long-form strings, short strings include escape sequences. Since invalid escape sequences count as parsing errors, these need to be read correctly by any proper Lua parser. Several aspects of string parsing are straightforward and need not be explained in detail:

- Unescaped newlines cause errors.
- If the string is quoted with single-quotes, double-quotes can be used in the contents unescaped, and vice-versa.
- Single-character escapes borrowed from C ("`\n`", "`\t`", etc.) and hexadecimal escapes from 00 to FF ("`\x0A`") are supported.
- Lua includes a special escape sequence "`\z`", which causes the parser to ignore all subsequent whitespace characters (including linebreaks) until the next non-whitespace character.

In C, an escape formed by a backslash and numeric digits is an octal escape, but in Lua, this is instead a decimal escape. Decimal escapes can be one to three digits long. "`\0`" through "`\255`" are valid, and any value above 255 is a syntax error. To parse this, the first character is grouped into one of the ranges 0-1, 2-2, or 3-9. Any decimal escape beginning with `\0` or `\1` cannot overflow, so any two digits are allowed to follow it. On the other hand, any decimal escape beginning with `\3` or `\9` will overflow iff it is three digits long. For the middle range, the second digit is read, and classified into ranges again in the same way (this time dividing at 5 instead of 2). If it is 5, this happens one final time for the third character.

Lua also supports escapes representing Unicode characters with specific codepoints; these are of the form "`\u{XYZ}`", where XYZ is a hexadecimal number (no leading "`0x`"). Any number between 0 and 7FFFFFFF is accepted, and can include any number of leading zeros. This is similar to the decimal-escape case explained above, and is parsed in approximately the same way.

3.2.1.8 Table Constructors

This subsystem parses table constructors, corresponding to the "`tableconstructor`" symbol in "The Complete Syntax of Lua." There is no specific input or output, and no entry/exit function is used.

A "table" in Lua is an unordered set of key-value pairs. In addition to scenarios where this sort of construct fits naturally, tables also serve as lists/arrays in this language, by using counting numbers as keys. This second usage is why the table constructor syntax allows "fields" (key-value pairs) lacking a key: fields lacking a key are automatically assigned one based on counting numbers.

There are three valid syntaxes for table fields: "[expression] = expression," "name = expression," and "expression".

If the first character in a field is "`[`", the "name = expression" production can be ruled out immediately. The "expression" production is still a possibility because of long-form strings: "`[[this is a string]]`" is a valid expression. Thus, the second character is then read to determine which of the two cases it is. From there, parsing is straightforward.

If the first character in a field is anything other than "`[`", parsing continues as an expression. The expression-parsing subsystem includes an optional feature that instructs it to inform the caller (via a stack symbol) whether the expression was a single name or anything else. In addition, the expression parser will always consume an "`=`" after an expression, since it may be the start of the "`==`" equality operator; if it turns out to not be this operator, this information is conveyed to the caller as well. Both of these facts are used here to distinguish the "name = expression" and "expression" cases.

3.2.1.9 Prefix Expressions and Variables

This subsystem parses the "`prefixexp`" and "`var`" symbols from "The Complete Syntax of Lua."

As defined in "The Complete Syntax of Lua" In "The Complete Syntax of Lua," the "`prefixexp`" and "`var`" symbols are defined as:

```
prefixexp ::= var | functioncall | '(' exp ')'
var ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name
functioncall ::= prefixexp args | prefixexp ':' Name args
```

The most notable fact here is that every variable is a valid prefix expression ("`prefixexp ::= var`"); thus, variables are a subset of prefix expressions. Prefix expressions additionally include function calls and expressions surrounded by parentheses. While the difference between variables and prefix expressions is not explained in the Lua documentation itself, it can be inferred from the rest of the grammar that "variables" refers to all valid L-values, and "prefix expressions" are constructions that are not themselves valid L-values, but can become one if suffixed with something like "[expression]" (indexing) or ".name" (syntactic sugar for indexing). Prefix expressions are a subset of valid R-values.

The EBNF definition is very precise and concise, but, due to the mutual recursion in the left halves of many of the productions, is not well-suited to creating a 2PDA parser. To do that, a reformulated definition is needed.

An Alternative Definition Here is an alternative way of defining these symbols, which progresses strictly from left to right.

A prefix expression or variable is a name or parenthesized expression, followed by zero or more of the following suffixes chained together:

- `[expression]`
- `. name`
- `: name (args)`
- `: name 'string argument'` or `[[string argument]]` or `[==[string argument ]==]`
- `: name {table constructor argument}`
- `(args)`
- `'string argument'` or `[[string argument]]` or `[==[string argument]==]`
- `{table constructor argument}`

If the final suffix is either of the first two ("`[expression]`", "`.name`"), then the sequence as a whole can be either a variable or a prefix expression, as desired. If the final suffix is any of the others, the sequence can only be a prefix expression. If the item is a bare name with no suffixes at all, then it can be either a variable or a prefix expression. If it is a bare parenthesized expression, it can only be a prefix expression.

Structure of Parser The alternative definition can be used to create a 2PDA parser.

The parser has two entry points: one where nothing has been consumed yet, and one where a name has already been consumed. The former is the "primary" entry point, and the latter is needed for parsing statements. Statements can begin with either a name or keyword, and the entire initial alphanumeric token must be read to determine which one it is. If it is a name, this prefix-expression-or-variable subsystem must be able to read the rest of the prospective L-value (since it is probably an assignment statement), with the initial name already consumed.

The parser has five exit points:

- Main exit: the full variable / prefix-expression was read.
- "Minus" exit: same as the above, but a trailing "`-`" was already consumed by the whitespace/comment parser.
- "Keyword" exit: the first token read was a keyword. This is used when parsing "return" statements, to distinguish e.g. "`function a() return x end`" from "`function a() return end`".
- "Period" exit: a "`.`" was consumed in the hopes that it would be a "`.name`" suffix, but it was found to be followed by another period. This forms the binary operator for concatenation, "`..`", and thus, the expression parser needs to know about this occurrence.
- "Colon" exit: a "`:`" was consumed in the hopes that it would be a "`:name`" suffix, but it was found to be followed by something else. This is used when reading statements: a function call may be followed by a label, which is surrounded by double-colons (e.g. "`::labelname::`").

The parser uses stack values to keep track of three variables: whether the construct parsed so far is valid as a variable, whether the construct parsed so far is a single bare name, and whether the most recently parsed suffix was a function call. The "single bare name" information is used by the expression parser, which in turn passes it on to the table-constructor parser, which uses it to identify the "name = expression" table field syntax. The function-call information is needed for the statement parser to determine if a lone variable-or-prefix-expression is valid as a statement or not (since expressions in Lua are generally not valid as statements, but function calls are specifically allowed).

A full walkthrough of the details of this parsing algorithm is given in Appendix A.

3.2.1.10 Expressions

This subsystem parses expressions, corresponding to the "`exp`" symbol in "The Complete Syntax of Lua."

This system has a large number of exit transitions, each with specific use cases in mind:

- Main exit: used in normal circumstances, when encountering an unexpected character that indicates the expression has ended.
- "end," "else," "elseif," and "until" exits: used if the first token read is one of these keywords, rather than the start of an actual expression. This feature is used when parsing return statements.
- "Semicolon" exit: used if the first token read is a semicolon. This feature is used when parsing return statements.
- "Right-paren" exit: used if the first token read is a right parenthesis ("`)`"). This feature is used when parsing function calls.

- “Equals” exit: used if the expression system consumes an “`=`” in the hopes of it being the start of the “`==`” equality binary operator, but some other character followed it instead.

- “Trailing name” exit: used if the expression system consumes a name in the hopes that it would be either the “`and`” or “`or`” keyword.

In addition, the system can optionally leave a value on the stack indicating if the entirety of the expression was a single name or not.

Expressions Beginning With Alphanumeric Characters If an expression begins with an alphabetic character, a name or keyword is read. If it is a name, parsing continues using the prefix-expression-or-variable subsystem. Said subsystem can optionally indicate if the entire prefix-expression-or-variable was a single bare name; this is used to determine and keep track of the same information with respect to the expression as a whole. If, instead, a keyword is found, it is compared to the values “`nil`”, “`true`” and “`false`” (which are valid as expressions on their own), the keywords “`function`” and “`not`” (which indicate function expressions and the unary “not” operator respectively), and “`end`”, “`else`”, “`elseif`”, and “`until`” (which trigger an immediate exit through the corresponding exit transitions).

If an expression begins with a digit, a numeric expression is parsed. Lua supports decimal and hexadecimal (prefixed with “`0x`”) numbers, both with optional fractional parts (with a “`.`”) and exponent markers (delimited by “`e`” for decimal numbers and “`p`” for hexadecimal numbers, each of which can also optionally be followed with a “`+`” or “`-`” before the actual exponent).

Expressions Beginning With Punctuation If an expression begins with a semicolon or right parenthesis, the system immediately exits through the corresponding exit point.

If an expression begins with a period, the next character is read; if it is another period, a "`...`" (known as the "vararg expression") is read, else a numeric expression is read. If it begins with a "`{`", a table constructor is read. If it begins with a "`(`", a prefix-expression-or-variable is read. If it begins with a "`-`", "`#`", or "`~`", this unary operator is consumed without affecting anything, since "`expression ::= unaryoperator expression`" is a valid rule in the Lua grammar. If it begins with a quote, a short string is read. If it begins with a "`[`", a long-form string is read.

Binary Operators and Looping Back Once one of the above constructs has been fully parsed, the subsystem attempts to read one of Lua's many binary operators. If it does so successfully, the expression parsing system restarts, to read the expression on the other side of the operator. If the next character is not the start of a binary operator, the expression parser exits.

3.2.1.11 Expression Lists

This subsystem parses expression lists, corresponding to the "`explist`" symbol in "The Complete Syntax of Lua."

The subsystem has a large number of exit transitions, but all are inherited directly from the expression parsing system, and simply delegate to it.

An expression list is a sequence of expressions separated by commas. Using the expression parsing system, this is very simple and straightforward to implement.

3.2.1.12 Statements

This subsystem reads a statement, corresponding to the "`stat`" symbol in "The Complete Syntax of Lua."

Statements Beginning With Alphanumeric Characters To parse a statement beginning with an alphanumeric character, the name-or-keyword subsystem is first used to identify whether the first token is a name or a keyword.

If the first token is a name, the statement could be either an assignment statement ("variablelist = expressionlist") or a function call. The prefix-expression-or-variable parser is applied, using an optional feature that checks if the prefix expression is a function call. Based on the result, the statement either ends here, or an "`=`" is read, followed by an expression list.

If the first token is a keyword, any of a variety of statement types may be read, depending on the specific keyword.

If an "`if`" is read, a value is pushed onto the stack indicating that an if-statement block is now active. An expression is read, followed by the "`then`" keyword, and then more statements. If an "`elseif`" is read while an if statement is active (according to the topmost stack value), it is treated the same way as an "`if`". If an "`else`" is read while an if statement is active, it is treated the same way again, except without the expression and "`then`". The stack value indicating an active "`if`" statement can be popped by reading the "`end`" keyword.

If a "`while`" is read, a stack value is pushed, an expression and "`do`" keyword is read, and statement parsing resumes. The stack value will be popped when an "`end`" keyword is read.

If a "`do`" is read, a stack value is pushed, and statement parsing resumes. The stack value will be popped when an "`end`" keyword is read.

Lua includes two types of for-loops, so if a "`for`" keyword is read, the parser needs to distinguish between them. The "numerical" for-loop uses the grammar rule "`stat ::= for Name '=' exp ',' exp [',' exp] do block end`", whereas the "generic" for-loop is defined as "`stat ::= for namelist in explist do block end`

". Both of these begin with a name after the "for" keyword, so that is parsed first. After this, either an "`=`" or a "`,`" is read. In either case, the rest of the loop is then read, in a similar way to the other types of block statements explained above.

If a "`repeat`" is read, a repeat-until loop is read. This is handled in the same manner as a while-loop or a do-end block, except that the "`until`" keyword at the end is followed by an additional expression.

If a "`function`" is read, a function statement is parsed. The keyword is followed by the function name, consisting of name tokens separated by "`.`"s, and optionally one final name separated by a "`:`". The function-body parser is then used to read the function body.

If a "`local`" is read, the statement could either be a local assignment statement ("`stat ::= local namelist ['=' explist]`") or a local function statement ("`stat ::= local function Name funcbody`"). This is easily disambiguated by reading the next token and seeing if it is the "`function`" keyword or not. After that, parsing continues in the same way as either an assignment statement or a function statement.

If a "`return`" is read, a return statement is read. A unique aspect about return statements in Lua is that they are syntactically required to be located at the end of a block. The PDA enforces this by reading the return statement's optional expression list and optional semicolon, followed by one of the keywords that ends a block ("`else`", "`elseif`", "`until`", "`end`"). Each of these keywords is then handled in the same way as if it had been encountered on its own.

If a "`break`" is read, nothing further is done, and parsing continues onto the next statement.

If a "`goto`" is read, a name is read following it, before continuing to the next statement.

If an "`end`" is read, and the value on top of the stack indicates that a statement containing another block is active (for "`do`", "`while`", "`if`", and "`for`" statements), the stack value is popped and statement parsing resumes. If the value on top of the stack indicates that a function body is being parsed, control returns to that subsystem instead.

Statements Beginning With Punctuation If a statement begins with "`;`", the character is read as a complete expression, since a semicolon is defined as a no-op statement in Lua. If it begins with a "`:`", a label ("`::labelname::`") is read. If it begins with "`(`", a prefix-expression-or-variable is read, and processed as either a function call or the start of an assignment statement.

3.2.1.13 Blocks

A block, corresponding to the "`block`" symbol in "The Complete Syntax of Lua," is essentially a sequence of statements (the "`stat`" symbol). This is implemented by parsing statements repeatedly until an "`end`" keyword is encountered.

3.2.1.14 Overall Entry Point

The overall entrypoint for the entire Lua parser begins by handling a special case described in the Lua documentation: "The first line in the file is ignored if it starts with a #." [2] After that, the rest of the file is a block, so the parser jumps directly to the block-parsing subsystem.

Chapter 4

EVALUATION

The Lua 2PDA was tested against two primary test suites: one developed specifically for the purpose, and the official test suite used by the reference implementation of Lua.

4.1 Custom Tests Based on Grammar

A custom test suite was constructed using the pytest unit testing framework [3], consisting of tests covering each part of the Lua syntax.

Test cases are grouped into over 25 functions, depending on the part of the syntax under test. Each test function includes an average of 13.7 unique test cases, and an average of 336.7 total test strings, using loops to automatically vary whitespace and other parameters. (This includes the numeral-expressions test function, which is an outlier that tests 5338 strings. If that function is not counted, the average is reduced to 144.4.) Many of the test cases contain invalid inputs, serving to show that the 2PDA not only accepts valid inputs but also rejects invalid ones.

Figure 4.1 (page 23) shows the specific breakdown of test cases by function/category.

The tests for numeral expressions, in particular, are comprehensive. Through deeply nested loops, every combination of the following is tested:

- Decimal or hexadecimal
- Leading zero or lack of one
- 0, 1, 2, or 3+ digits long

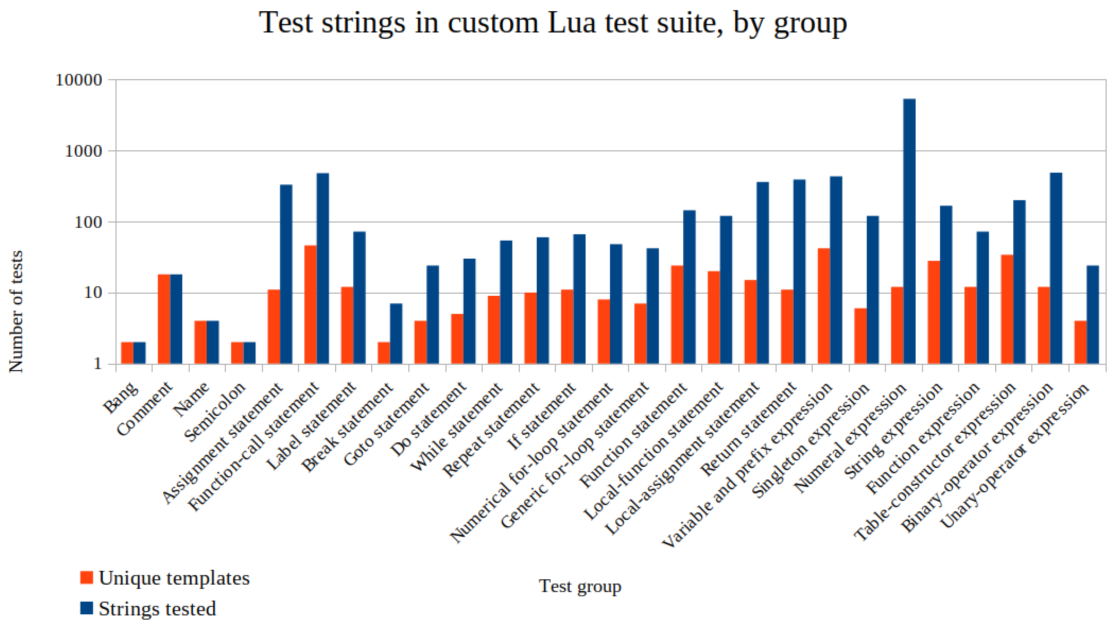

Figure 4.1: Amounts of strings in each test group in the custom test suite, in a log scale. "Unique templates" is the number of unique test-case string literals in the source code; "strings tested" is the total number of strings tested at runtime. Strings tested generally outnumbers "unique templates" due to the use of loops to automatically test different cases.

- Fractional part or lack of one
- Exponent part or lack of one
- "+", "-", or no sign in the exponent
- Uppercase / lowercase (for hexadecimal)

The Lua 2PDA passes 100% of the custom test suite.

All together, these tests cover every individual portion of the Lua grammar, as well as regression tests for bugs discovered in the 2PDA. However, to test interactions between different syntax constructs, a different type of test suite is needed.

4.2 Lua Test Suite

The 2PDA was also evaluated against the official Lua test suite. This was chosen as an example of code that is closer to real-life usage and which can reasonably be expected to cover most or all aspects of the language syntax. Since these tests are just loose Lua code files, a thin pytest wrapper was written to load them into the 2PDA.

The Lua 2PDA recognizes 100% of the files in the official Lua test suite.

Chapter 5

RELATED WORK

There exists a well-known algorithm for converting grammars to pushdown automata, taught in introductory theoretical computer science classes. However, the output of this algorithm is not necessarily a deterministic pushdown automaton, and determinism is required for the applications this Lua 2PDA was designed for. In addition, an automatically generated 2PDA would not include the copious comments and explanations that a manually written one does, which could make it more difficult to use for this sort of research. However, it would have the benefits of being more easily and more quickly applied to a larger number of languages. This option should be explored further.

There also exists more advanced automatic parser generation software that accepts grammars as input; a prominent example of this is ANTLR [5]. ANTLR was considered as an alternative to this project; however, its output is actual source code for a parser, rather than a formal pushdown automaton. Thus, it is not directly applicable to the use cases the author has in mind.

Chapter 6

CONCLUSION

This project introduced a new two-way deterministic finite pushdown automaton implementing the grammar of the Lua language, as well as a test suite to ensure its accuracy with a high degree of confidence. Not only did the 2PDA fully pass its test suite, it also fully passed the official test suite used by the reference implementation of Lua itself. This 2PDA can potentially be used in future research regarding identifying the language and current parsing state of a black-box parser.

REFERENCES

[1] “Lua implementations”, https://luajit.org/luajit.html (2018).

[2] Ierusalimschy, R., L. H. de Figueiredo and W. Celes, “Lua 5.3 reference manual”, https://www.lua.org/manual/5.3/manual.html (2015–2018).

[3] Krekel, H., “pytest: helps you write better programs”, https://docs.pytest.org/en/latest/ (2004–2020).

[4] Pall, M., “Luajit”, https://luajit.org/luajit.html (2005–2020).

[5] Parr, T., “About the antlr parser generator”, https://www.antlr.org/about.html (2014).

APPENDIX A

ALGORITHM USED FOR THE PREFIX-EXPRESSION/VARIABLE PARSING SYSTEM

This appendix provides a conceptual description of the algorithm used by the 2PDA to read a prefix expression or variable, and determine which of those it has read. The system has two entry points, five exits, and three output variables. Because the algorithm's structure does not closely align with those of the productions of the "prefixexp" and "var" symbols in "The Complete Syntax of Lua," a high-level algorithm explanation is warranted.

Among other details, descriptions of when to parse optional whitespace are omitted here for brevity and clarity.

A.1 Entry Point 1 (Nothing Consumed Yet)

- Inspect the first character.
- If it looks like a name or keyword...
  - Read the name or keyword.
  - If it is a name...
    - (This is equivalent to entry point 2 now.)
    - Initialize stack values to "`prefixexp_or_var`", "`only_name`", and "`not_function_call`", and go to "Read Next Part."
  - If it is a keyword...
    - Take the appropriate exit ("keyword").
- If it is a "`(`"...
  - This is "`(` expression `)`".
  - Read an expression, and then "`)`".
  - Initialize stack values to "`prefixexp`", "`not_only_name`", and "`not_function_call`", and go to "Read Next Part."

A.2 Entry Point 2 (A Name Has Already Been Consumed)

- Initialize stack values to "`prefixexp_or_var`", "`only_name`", and "`not_function_call`", and go to "Read Next Part."

A.3 "Read Next Part"

- Inspect the next character.
- If it is a "`(`" or "`{`" or single quote or double quote...

  - This is a function call.
  - Read arguments (expression list, or table, or string).
  - Replace stack values with “`prefixexp`”, “`not_only_name`”, and “`function_call`”, and go to “Read Next Part.”

- If it is a “`[`”...
  - This is either a function call (long-form string argument) or indexing.
  - Consume it, and check the next character.
  - If it is a “`[`” or “`=`”...
    - Handle as a function call starting with a long-form string.
    - Replace stack values with “`prefixexp`”, “`not_only_name`”, and “`function_call`”, and go to “Read Next Part.”
  - If it is anything else...
    - This is indexing.
    - Read an expression and then “`]`”.
    - Replace stack values with “`prefixexp_or_var`”, “`not_only_name`”, and “`not_function_call`”, and go to “Read Next Part.”

- If it is a “`.`”...
  - If the next character is another “`.`”...
    - This might be the binary concatenation operator “`..`”, which should be handled by the caller.
    - Take the “period” exit.
  - If it is anything else...
    - This is member lookup.
    - Read a name.
    - Replace stack values with “`prefixexp_or_var`”, “`not_only_name`”, and “`not_function_call`”, and go to “Read Next Part.”

- If it is a “`:`”...
  - If the next character is another “`:`”...
    - This might be the start of a label (“`::labelname::`”), which should be handled by the caller.
    - Take the “colon” exit.
  - If it is anything else...
    - This is a function call.
    - Read a name, and then handle as a function call.
    - Replace stack values with “`prefixexp`”, “`not_only_name`”, and “`function_call`”, and go to “Read Next Part.”

- If it is anything else...
  - This is the end of the prefix-expression-or-variable. Take either the main or “minus” exit, depending on the result of the final whitespace reading.