

Parser for LUA

Rayudu Singaraju Roll no: 37

Akhil Vanukuri Roll no: 38

Semester VI , Section C
Batch 2

Department of Computer Science & Engineering



MANIPAL
INSTITUTE OF TECHNOLOGY
A Constituent Institute of Manipal University, Manipal

CONTENTS

1. Abstract
2. Objective
3. Introduction
 - 3.1. Grammar
 - 3.2. Language used for implementation.
 - 3.3. Type of parser
4. Methodology
5. Code
 - 5.1. Flex Code
 - 5.2 Bison Code
 - 5.3 User Documentation
6. Results
7. References

Abstract: There's a vast variety of high level languages available in the market and every such language needs an interpreter or a compiler to make it machine understandable. Parsing and Token Generation are few important stages in the construction of a parser or a compiler. The goals of this project are : (1) Understanding and analyzing the above stated stages of compiler design in general (2) Construction of a parser for the programming language Lua.

Objective: (1) Understand and Analyze the *Lexical Analysis Phase*: To understand how tokens are generated and passed on to the later stages by creating a lexical analyzer for Lua. (2) Understand and Analyze the *Syntax Analysis Phase*: To understand how the grammar is generated and how a language is created using that grammar by taking an example of Lua. (3) Discuss Limitations of the implementation methods used.

Introduction: [1] *Lua* is an extension programming language. Being an extension language, *Lua* has no notion of a "main" program. The *Lua* distribution includes a sample host program called *lua*, which uses the *Lua* library to offer a complete, stand-alone *Lua* interpreter.

[2] A *compiler* is a program that takes as input a program written in one language and translates it into a functionally equivalent program in the target language. As it translates, a compiler also reports errors and warnings to help the programmer make corrections to the source, so the translation can be completed.

[2] *Lexical Analysis or Scanning*: The stream of characters making up a source program is read from left to right and grouped into tokens, which are sequences of characters that have a collective meaning. Examples of tokens are identifiers (user-defined names), reserved words, integers, doubles or floats, delimiters, operators, and special symbols.

[2] *Syntax Analysis or Parsing*: The tokens found during scanning are grouped together using a context-free grammar. A grammar is a set of rules that define valid structures in the programming language. Each token is associated with a specific rule, and grouped together accordingly. This process is called parsing.

[1] *Grammar*: Grammar for this project has been given in the above mentioned reference in BNF notation, the grammar has been modified a bit and has been re-written in CFG notation.

Grammar:

chunk --> (stat (";")?)* (last_stat (";")?)?

block --> *chunk*

stat --> var_list "=" exp_list |
function_call |
do block end |
while exp do block end |
repeat block until exp |

```

    if exp then block (elseif exp then block)* (else block)? end |
    for id "=" exp "," exp ("," exp)? do block end |
    for name_list in exp_list do block end |
    function func_name func_body |
    local function id func_body |
    local name_list ("=" exp_list)?

last_stat -->    return (exp_list)? | break

func_name -->    id ( "." id)* ( ":" id)?

var_list  -->    var ( "," var)*

var -->          id | prefix_exp "[" exp "]" | prefix_exp "." id

name_list -->    id ( "," id)*

exp_list -->    (exp ",")* exp

exp -->          nil | false | true | Number | String | "..." | function |
                prefix_exp | table_constructor | '(' exp bin_op exp ')' | un_op exp

prefix_exp -->   var | function_call | "(" exp ")"

function_call --> prefix_exp args | prefix_exp ":" id args

args -->          "(" (exp_list)? ")" | table_constructor | String

function -->      function func_body

func_body -->     "(" (par_list)? ")" block end

par_list -->      name_list ( "," "..." )? | "..."

table_constructor --> "{" (field_list)? "}"

field_list -->    field (field_sep field)* (field_sep)?

field -->         "[" exp "]" "=" exp | id "=" exp | exp

field_sep -->     "," | ";"

bin_op -->        "+" | "*" | "/" | "^" | "%" | ".." |
                "<" | "<=" | ">" | ">=" | "==" | "~=" |
                and | or

un_op -->         "-" | not | "#"

id -->            [a-zA-z][a-zA-z_0-9]*

String -->        \"(\\.|[^\"])*\"

digit -->         [0-9]

```

Changes from the Original Grammar:

1.Only integers are used.

2.An extra pair of brackets have been added to exp op exp to resolve few conflicts.

Language used for Implementation:

Lexical Analysis and Token generation : Flex

Parsing : Bison

Flex is a fast lexical analyzer generator[4].It takes your specification and generates a combined NFA to recognize all your patterns, converts it to an equivalent DFA, minimizes the automaton as much as possible, and generates C code that will implement it.

[3]*Bison* is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar.

Type of parser used:Bottom-up parser

The *Bison* parser is a bottom-up parser[3]. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.

Methodology:

First,we decided on which parts of the grammar should be converted into tokens ,then we created a flex file using regular expression like notation to denote the lexemes to be generated and then we assigned token values to the lexemes.

Then we linked this file to the bison file and made it such that bison takes token generated by the flex file as inputs.

We also wrote a bison file which has the syntax rules,that checks if parsing is happening without any error.

If any error occurs,then it displayed using the reference taken from a baby C compiler code [5]Then a Symbol Table was constructed that could be used to store user defined identifiers and Functions by defining a structure (sym_tab) as shown in the code below.

Once the first running version of the code was done ,it has been checked on different inputs and errors have been rectified in an incremental fashion and the other group in C1 that was working on lua helped us with a string input error which we were not aware of .

Code:

Flex Code:

```
%{
    #include "lua.tab.h"
    #include <string.h>
    #include <stdio.h>
    #include <stdlib.h>

    int col_no = 1;
    int line_no = 1;
    int counter = 0;

    int check = 0;
    int s_check = 0;

    typedef struct node
    {
        char name[50];
        int type;
        int col;
        int row;
        int scope;
    }sym_tab;

    sym_tab symbol_table[100];

    int Search_Symbol(char* str)
    {
        for(int i = 0 ; i < counter ; i++)
        {
            if(symbol_table[i].type == check && strcmp(symbol_table[i].name,str) == 0)
            {
                return 1;
            }
        }
        return 0;
    }
}%}

%option yylineno

%%
"$" {printf(" %s ",yytext);col_no += strlen(yytext);return DOLLAR;}
"--"(.)*"\n" {col_no = 0 ; line_no++;}
("\t"|" "|"\n") {if(strcmp(yytext,"\n")==0){col_no = 0;line_no++;} else
if(strcmp(yytext,"\t")==0){col_no += 4;}else col_no += strlen(yytext);}
\"(\\.|[ ^\"\\])*\" {printf(" %s ",yytext);col_no += strlen(yytext);return STRING;}
and {printf(" %s ",yytext);col_no += strlen(yytext);return AND;}
break {printf(" %s ",yytext);col_no += strlen(yytext);return BREAK;}
do {printf(" %s ",yytext);col_no += strlen(yytext);return DO;}
else {printf(" %s ",yytext);col_no += strlen(yytext);return ELSE;}
```

```

elseif {printf(" %s ",yytext);col_no += strlen(yytext);return ELSEIF;}
end {printf(" %s ",yytext);col_no += strlen(yytext); return END;}
false {printf(" %s ",yytext);col_no += strlen(yytext);return FALSE;}
for {printf(" %s ",yytext);col_no += strlen(yytext);return FOR;}
function {printf(" %s ",yytext);col_no += strlen(yytext); check = 1;return FUNCTION;}
if {printf(" %s ",yytext);col_no += strlen(yytext);return IF;}
in {printf(" %s ",yytext);col_no += strlen(yytext);return IN;}
local {printf(" %s ",yytext);col_no += strlen(yytext); s_check = 1;return LOCAL;}
nil {printf(" %s ",yytext);col_no += strlen(yytext);return NIL;}
not {printf(" %s ",yytext);col_no += strlen(yytext);return NOT;}
or {printf(" %s ",yytext);col_no += strlen(yytext);return OR;}
repeat {printf(" %s ",yytext);col_no += strlen(yytext);return REPEAT;}
return {printf(" %s ",yytext);col_no += strlen(yytext);return RETURN;}
then {printf(" %s ",yytext);col_no += strlen(yytext);return THEN;}
true {printf(" %s ",yytext);col_no += strlen(yytext);return TRUE;}
until {printf(" %s ",yytext);col_no += strlen(yytext);return UNTIL;}
while {printf(" %s ",yytext);col_no += strlen(yytext);return WHILE;}
("#") {printf(" %s ",yytext);col_no += strlen(yytext);return HASH;}
("+","*","/","^","%","|","<","=",">",">=","|","~=") {printf(" %s ",yytext);col_no +=
strlen(yytext);return BINOP;}
";" {printf(" %s ",yytext);col_no += strlen(yytext);return SEMICOLON;}
"," {printf(" %s ",yytext);col_no += strlen(yytext);return COMMA;}
"=" {printf(" %s ",yytext);col_no += strlen(yytext);return EQUALS;}
"(" {printf(" %s ",yytext);col_no += strlen(yytext);return LB;}
")" {printf(" %s ",yytext);col_no += strlen(yytext);return RB;}
"{" {printf(" %s ",yytext);col_no += strlen(yytext);return LFB;}
"}" {printf(" %s ",yytext);col_no += strlen(yytext);return RFB;}
"[" {printf(" %s ",yytext);col_no += strlen(yytext);return LSB;}
"]" {printf(" %s ",yytext);col_no += strlen(yytext);return RSB;}
":" {printf(" %s ",yytext);col_no += strlen(yytext);return COLON;}
"..." {printf(" %s ",yytext);col_no += strlen(yytext);return VARDOT;}
"." {printf(" %s ",yytext);col_no += strlen(yytext);return DOT;}
"-" {printf(" %s ",yytext);col_no += strlen(yytext);return MINUS;}

```

```

[A-Za-z_][0-9A-Za-z_]* {
    printf(" %s ",yytext);
    if(!Search_Symbol(yytext))
    {
        symbol_table[counter].col = col_no;
        symbol_table[counter].row = line_no;
        symbol_table[counter].type = check;
        symbol_table[counter].scope = s_check;

        check = 0;
        s_check = 0;

        strcpy(symbol_table[counter].name,yytext);
        counter++;
    }
    col_no += strlen(yytext);
    return ID;
}

```

```

[0-9]+ {printf(" %s ",yytext);col_no += strlen(yytext);return NUM;}
(.) {col_no += strlen(yytext);}

```

```
%%
int yywrap()
{
    return 1;
}
```

Bison Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
extern FILE* yyin;
int yyerror();
int yylex();
extern char yytext[];
extern int line_no;
extern int col_no;

typedef struct node
{
    char name[50];
    int type;
    int col;
    int row;
    int scope;
}sym_tab;

extern sym_tab symbol_table[];
extern int counter;

//symbol table considers only user defined functions as the others are already assumed to be
designed and for now it takes them as identifiers
//All the Table Constructs Should be Added with Flower brackets
//Expressions of type exp op exp should be given in ( exp op exp)s
%}

#define parse.error verbose
%token AND BREAK DO ELSE ELSEIF END FALSE FOR FUNCTION IF IN LOCAL NIL NOT OR
REPEAT RETURN THEN TRUE UNTIL WHILE
%token HASH BINOP EQUALS MINUS
%token LB RB LFB RFB LSB RSB
%token SEMICOLON COMMA COLON DOT VARDOT
%token ID NUM STRING
```


%token DOLLAR

%glr-parser

%start s

%%

s : *block DOLLAR*;

block : *chunk* ;

chunk : *star_stat last_stat_opt*
/;

star_stat : *star_stat stat opt_semicolon*
/;

last_stat_opt : *last_stat opt_semicolon*
/;

opt_semicolon : *SEMICOLON*
/;

stat : *var_list EQUALS exp_list*
/ *function_call*
/ *DO block END*
/ *WHILE exp DO block END*
/ *REPEAT block UNTIL exp*
/ *IF exp THEN block elseif_star else_opt END*
/ *FOR ID EQUALS exp COMMA exp comma_exp_opt DO block END*
/ *FOR name_list IN exp_list DO block END*
/ *FUNCTION func_name func_body*
/ *LOCAL FUNCTION ID func_body*
/ *LOCAL name_list opt_equals*;

comma_exp_opt : *COMMA exp*
/;

elseif_star : *elseif_star ELSEIF exp THEN block*
/;

```

else_opt      : ELSE block
               /;

opt_equals    : EQUALS exp_list
               /;

last_stat     : RETURN exp_list_opt
               /BREAK;

exp_list_opt  : exp_list
               /;

func_name     : ID dot_id_star colon_id_opt {printf("\nDone");};

dot_id_star   : dot_id_star DOT ID
               /;

colon_id_opt  : COLON ID
               /;

var_list      : var var_star;

var_star      : var_star COMMA var
               /;

var           : ID
               /prefix_exp LSB exp RSB
               /prefix_exp DOT ID {printf("\nDone");};

name_list     : ID comma_id_star;

comma_id_star : comma_id_star COMMA ID
               /;

exp_list      : exp_comma_star exp;

exp_comma_star : exp_comma_star exp COMMA
               /;

exp           : NIL
               /FALSE

```

	TRUE
	NUM
	STRING
	VARDOT
	func
	prefix_exp
	table_const
	LB exp BINOP exp RB
	LB exp OR exp RB
	LB exp AND exp RB
	unop exp;
prefix_exp	: var function_call LB exp RB;
function_call	: prefix_exp args prefix_exp COLON ID args
args	: LB exp_list_opt RB STRING table_const;
func	: FUNCTION func_body;
func_body	: LB par_list_opt RB block END;
par_list_opt	: par_list ;
par_list	: name_list comma_vardot_o VARDOT;
comma_vardot_o	: COMMA VARDOT ;
table_const	: LFB field_list_opt RFB;
field_list_opt	: field_list ;
field_list	: field f_sep_f_star field_sep_op

```
f_sep_f_star      : f_sep_f_star field_sep field
                    |;
```

```
field_sep_op      : field_sep
                    |;
```

```
field             : LSB exp RSB EQUALS exp
                    |ID EQUALS exp
                    |exp;
```

```
field_sep         : COMMA
                    |SEMICOLON;
```

```
unop              : MINUS
                    | NOT
                    | HASH;
```

```
%%
```

```
int yyerror(const char *str)
{
    printf("\ninput.lua : ERROR %d.%d %s\n",line_no,col_no,str);
    return 1;
}
```

```
void main ()
{
    yyin = fopen("in","r");
    if(yyparse())
    {
        printf("FAILURE\n");
        exit(0);
    }
    printf("\nSUCCESS\n");

    for(int i = 0 ; i < counter ; i++)
    {

        printf("Name : %s\tLine : %d\tColumn : %d\t",symbol_table[i].name,symbol_table[i].row,symbol_table[i].col);

        if(symbol_table[i].scope == 1)
```

```

        printf("Local\t");
    else
        printf("Global\t");

    if(symbol_table[i].type == 1)
        printf("Function\n");
    else
        printf("Identifier\n");
    }
}

```

Results:

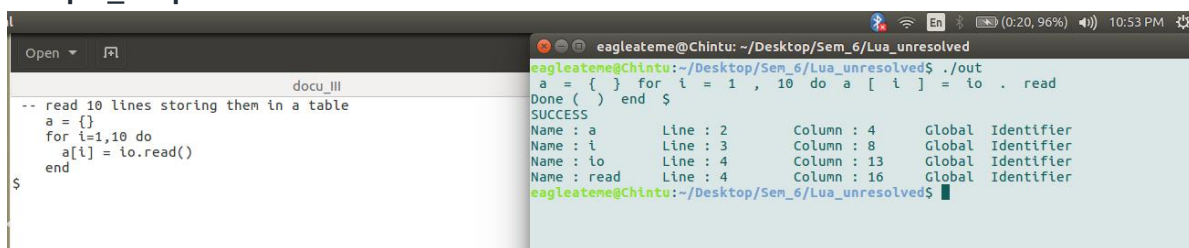
Sample_Input:

```

-- read 10 lines storing them in a table
a = {}
for i=1,10 do
    a[i] = io.read()
end

```

Sample_Output:



```

eagleateme@Chintu: ~/Desktop/Sem_6/Lua_unresolved
eagleateme@Chintu:~/Desktop/Sem_6/Lua_unresolved$ ./out
a = { } for i = 1 , 10 do a [ i ] = io . read
Done ( ) end $
SUCCESS
Name : a      Line : 2      Column : 4      Global Identifier
Name : i      Line : 3      Column : 8      Global Identifier
Name : io     Line : 4      Column : 13     Global Identifier
Name : read   Line : 4      Column : 16     Global Identifier
eagleateme@Chintu:~/Desktop/Sem_6/Lua_unresolved$

```

This Code has been tested against all the code that has been attached at the end of the document.

User Documentation:

LUA PARSER

Requirements:

Understanding of LR automata to interpret lua.output file

Note:

- 1.Symbol table considers only user defined functions as the others are already assumed to be designed and for now it takes them as identifiers
- 2.Expressions of type exp op exp should be given in (exp op exp)

Usage:

When Running for the first time,compile the files using the below stated commands in the terminal/cmd

```
> flex -l lua.l
```

```
> bison -vd lua.y
```

```
> gcc lex.yy.c lua.tab.c -o out
```

load (in) the file in the folder with the test code that you want to check.

run it using ./out command

Interpretation of the output:

lua.output file can be used to see default action chosen in case of conflicts.

The output first displays all the tokens and then displays the Symbol table only on successful parsing otherwise an error message is displayed which is self explainable.

Debugging Help:

Functions:

```
void LibFuc();
```

```
""This Function loads the system defined functions into the symbol table""
```

```
int Search_Symbol(char* str)
```

```
"" This Function searches for the identifier with the given name and returns 1 if it's found else 0,the input to the function is a string""
```

```
int yywrap();
```

```
"" Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. ""
```

```
int yyerror(char* str);
```

""This Function returns 1 on encountering an error while parsing and also displays the error""

```
int yylex();
```

""The lexical analyzer function, yylex , recognizes tokens from the input stream and returns them to the parser ""

```
int yyparse();
```

""This function reads tokens, executes actions, and ultimately returns when it encounters end-of-input or an unrecoverable syntax error""

Structures:

snippet of code:

```
typedef struct node
{
    char name[50];
    int type;
    int col;
    int row;
    int scope;
}sym_tab;
```

This data structure has been used to define a single entry into the symbol table and contains:

type : it stores 1 if it's a function and 0 if it's an identifier

col : column number of the stored identifier.

row : row number of the stored identifier.

scope : it stores 0 if it is a Global Variable otherwise stores 1 to indicate that it is a local variable.

Limitations:

As this is a subset of the actual lua grammar, we have tried to be as faithful as possible to the original Lua grammar but in order to reduce conflicts, we have made some changes to the original grammar

1. The `exp bin_op exp` production has been modified to `LB exp bin_op exp RB`. So, if `a = 1+2` is there, it should be written as `(a+b)` in the grammar.

2. Minus has been removed from the binary operation list as it has been creating a conflict with Unary minus, so now every expression of the form `a-b` has to be represented as `(a - b)`

3. Some features of the original Lua grammar like strings being represented as `[[string]]` have been dropped

4. Variable args have been dropped from using anywhere else other than function definition.
 5. Only Integers can be represented
 6. Symbol Table does not give a proper output for non-user defined functions as it has been assumed that such functions are entered into the symbol table at the start itself.
 7. Total No. of Shift/Reduce Conflicts : 02
 8. Total No. of Reduce/Reduce Conflicts : 15
- The Default in case of Conflicts is directly decided by Bison and is compatible with the syntax of the language.
- The conflicts are because of the fact that bison is LALR(1) parser and the language (lua) has some features that cannot be represented using a LALR(1) grammar.

References:

- [1] <http://www.lua.org/manual/5.1/manual.html>
- [2] <https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/020%20CS143%20Course%20Overview.pdf>
- [3] <https://www.gnu.org/software/bison/#content>
- [4] <https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/050%20Flex%20In%20A%20Nutshell.pdf>
- [5] <https://github.com/Wilfred/babyc>
- [6] O'REILLY Flex and Bison by John Levine