lua-parser Grammar

Patterns highlighted in orange mean @symb(patt), and those in green mean @kw(patt). Slight adjustments have been made to the grammar for clarity (mostly inlining).

```
Lua           <- Shebang? Skip Block !.
Shebang       <- '#' (!'\n' .)* '\n'

Block         <- StatList RetStat?
StatList      <- (';' / Stat)*
Stat          <- IfStat / WhileStat / DoStat / ForStat / RepeatStat / FuncStat
               / LocalStat / LabelStat / BreakStat / GoToStat / ExprStat

IfStat        <- 'if' Expr 'then' Block
                ('elseif' Expr 'then' Block)*
                ('else' Block)? 'end'

DoStat        <- 'do' Block 'end'
WhileStat     <- 'while' Expr 'do' Block 'end'
RepeatStat    <- 'repeat' Block 'until' Expr

ForStat       <- 'for' (ForNum / ForGen) 'end'
ForNum        <- Id '=' Expr ',' Expr (',' Expr)? ForBody
ForGen        <- NameList 'in' ExpList ForBody
ForBody       <- 'do' Block

LocalStat     <- 'local' (LocalFunc / LocalAssign)
LocalFunc     <- 'function' Id FuncBody
LocalAssign   <- NameList ('=' ExpList)?

FuncStat      <- 'function' FuncName FuncBody
FuncName      <- Id ('.' @token(Name))* (':' @token(Name))?
FuncBody      <- '(' ParList? ')' Block 'end'
ParList       <- NameList (',' '...')? / '...'

ExprStat      <- SuffixedExp Assignment / SuffixedExp
Assignment    <- (',' SuffixedExp)* '=' ExpList

LabelStat     <- '::' @token(Name) '::'
BreakStat     <- 'break'
GoToStat      <- 'goto' @token(Name)
RetStat       <- 'return' (Expr (',' Expr)*)? ';'?

@sepby1(p, sep) <- p (sep p)*
NameList      <- @sepby1(Id, ',')
ExpList       <- @sepby1(Expr, ',')
```

```
@chainl1(p, op) <- p (op p)*
Expr            <- SubExpr_1
SubExpr_1       <- @chainl1(SubExpr_2, 'or')
SubExpr_2       <- @chainl1(SubExpr_3, 'and')
SubExpr_3       <- @chainl1(SubExpr_4, '~=' / '==' / '<=' / '>=' / '<' / '>')
SubExpr_4       <- @chainl1(SubExpr_5, '|')
SubExpr_5       <- @chainl1(SubExpr_6, '~')
SubExpr_6       <- @chainl1(SubExpr_7, '&')
SubExpr_7       <- @chainl1(SubExpr_8, '<<' / '>>')
SubExpr_8       <- SubExpr_9 ('..' SubExpr_8)?
SubExpr_9       <- @chainl1(SubExpr_10, '+' / '-')
SubExpr_10      <- @chainl1(SubExpr_11, '*' / '//' / '/' / '%')
SubExpr_11      <- ('not' / '-' / '#' / '~') SubExpr_11 / SubExpr_12
SubExpr_12      <- SimpleExp ('^' SubExpr_11)?

SimpleExp       <- @token(Number) / @token(String) / 'nil' / 'false' / 'true' / '...'
                 / FunctionDef / Constructor / SuffixedExp

SuffixedExp     <- PrimaryExp ('.' @token(Name) / '[' Expr ']' / ':' @token(Name) FuncArgs
/ FuncArgs)*
PrimaryExp      <- Id / '(' Expr ')'

FunctionDef     <- 'function' FuncBody
FuncArgs        <- '(' (Expr (',' Expr)*)? ')' / Constructor / @token(String)

Constructor     <- '{' FieldList '}'
FieldList       <- (Field (FieldSep Field)* FieldSep?)?
Field           <- '[' Expr ']' '=' Expr / @token(Name) '=' Expr / Expr
FieldSep        <- ',' / ';'

@kw(p)          <- @token(p !idRest)
@symb(p)        <- @token(p)
@token(p)       <- p Skip

Skip            <- (Space / Comment)*
Space           <- space+
Comment         <- '--' LongString / '--' (!'\n' .)*

Id              <- @token(Name)
Name            <- !Reserved Identifier !idRest Skip
Reserved        <- Keywords !idRest
Identifier      <- idStart idRest*

idStart         <- alpha / '_'
idRest          <- alnum / '_'
Keywords        <- 'and' / 'break' / 'do' / 'elseif' / 'else' / 'end'
                 / 'false' / 'for' / 'function' / 'goto' / 'if' / 'in' / 'local'
                 / 'nil' / 'not' / 'or' / 'repeat' / 'return' / 'then' / 'true'
```

```
               / 'until' / 'while'

Number       <- Hex / Float / Int
Hex          <- ('0x' / '0X') xdigit+
Float        <- (digit+ '.' digit* / '.' digit+) Expo? / digit+ Expo
Expo         <- [eE] [+-]? digit+
Int          <- digit+

String       <- LongString / ShortString
Shor@token(String)  <- '"' ('\\' . / !'"' .)* '"' / '\'' ('\\' . / !'\'' .)* '\''
LongString   <- Open (!Close .)* Close  -- Close must have same # of '='s as Open
Open         <- '[' Equals '[' '\n'?
Close        <- ']' Equals ']'
Equals       <- '='*

alnum        <- <lpeg.alnum>   -- [A-Za-z0-9]
alpha        <- <lpeg.alpha>   -- [A-Za-z]
digit        <- <lpeg.digit>   -- [0-9]
space        <- <lpeg.space>   -- [ \t\n]
Xdigit       <- <lpeg.xdigit>  -- [0-9a-fA-F]
```

Current Status of Error Reporting in lua-parser

Error Detection

- Tracks the farthest failing position during parsing
- Also checks some additional errors like breaks outside of a loop

Error Reporting

- Prints the unexpected token encountered
- Provides the line and column number of the error
- Shows a list of possible tokens expected

Error Recovery

- None - stops on the first error

Sample

```
-- missing operand/expression on right side of +
parse("print(1+)", "code.lua")
```

```
code.lua:1:9: syntax error, unexpected ')', expecting '(', 'Name', '{',
'function', '...', 'true', 'false', 'nil', 'String', 'Number', '~', '#', '-', 'not'
```

Annotated lua-parser Grammar (without Recovery)

Significant changes are highlighted in red and splits in violet. Minor changes & renames are not highlighted.
(pattern)^{label} means that the label is thrown upon failure of the pattern (the parentheses are optional;
***note:** this differs from the previous notation which used square brackets).*

```
Lua           <- Shebang? Skip Block !.^ErrExtra
Shebang       <- '#!' (!'\n' .)*

Block         <- Stat* RetStat?
Stat          <- IfStat / WhileStat / DoStat / ForStat / RepeatStat
               / LocalStat / FuncStat / BreakStat / LabelStat / GoToStat
               / FuncCall / Assignment / ';' / !BlockEnd ErrInvalidStat
BlockEnd      <- 'return' / 'end' / 'elseif' / 'else' / 'until' / !.

IfStat        <- IfPart (ElseIfPart)* (ElsePart)? 'end'^ErrEndIf
IfPart        <- 'if' Expr^ErrExprIf 'then'^ErrThenIf Block
ElseIfPart    <- 'elseif' Expr^ErrExprEIf 'then'^ErrThenEIf Block
ElsePart      <- 'else' Block

DoStat        <- 'do' Block 'end'^ErrEndDo
WhileStat     <- 'while' Expr^ErrExprWhile 'do'^ErrDoWhile Block 'end'^ErrEndWhile
RepeatStat    <- 'repeat' Block 'until'^ErrUntilRep Expr^ErrExprRep

ForStat       <- 'for' (ForNum / ForIn)^ErrForRange 'end'^ErrEndFor
ForNum        <- Name '=' NumRange ForBody
NumRange      <- Expr^ErrExprFor1 ','^ErrCommaFor Expr^ErrExprFor2 (',' Expr^ErrExprFor3)?
ForIn         <- NameList 'in'^ErrInFor ExprList^ErrEListFor ForBody
ForBody       <- 'do'^ErrDoFor Block

LocalStat     <- 'local' (LocalFunc / LocalAssign)^ErrDefLocal
LocalFunc     <- 'function' Name^ErrNameLFunc FuncBody
LocalAssign   <- NameList ('=' ExprList^ErrEListLAssign)?
Assignment    <- VarList '=' ExprList^ErrEListAssign

FuncStat      <- 'function' FuncName^ErrFuncName FuncBody
FuncName      <- Name ('.' Name^ErrNameFunc1)* (':' Name^ErrNameFunc2)?
FuncBody      <- FuncParams Block 'end'^ErrEndFunc
FuncParams    <- '('^ErrOParenPList ParList? ')'^ErrCParenPList
ParList       <- NameList (',' '...'^ErrParList)? / '...'

LabelStat     <- '::' Name^ErrLabel '::'^ErrCloseLabel
GoToStat      <- 'goto' Name^ErrGoto
BreakStat     <- 'break'
RetStat       <- 'return' @commaSep(Expr, ErrRetList)? ';'?
```

```
@commaSep(patt, label)     <- @sepBy(patt, ",", label)
@chainOp(patt, op, label)  <- @sepBy(patt, op, label)
@sepBy(patt, sep, label)   <- p (sep p^label)*   -- label is optional

NameList      <- @commaSep(Name)
VarList       <- @commaSep(VarExpr, ErrVarList)
ExprList      <- @commaSep(Expr, ErrExprList)

Expr          <- OrExpr
OrExpr        <- @chainOp(AndExpr, 'or', ErrOrExpr)
AndExpr       <- @chainOp(RelExpr, 'and', ErrAndExpr)
RelExpr       <- @chainOp(BOrExpr, '~=' / '==' / '<=' / '>=' / '<' / '>',
                         ErrRelExpr)
BOrExpr       <- @chainOp(BXorExpr, '|', ErrBOrExpr)
BXorExpr      <- @chainOp(BAndExpr, '~' !'=', ErrBXorExpr)
BAndExpr      <- @chainOp(ShiftExpr, '&', ErrBAndExpr)
ShiftExpr     <- @chainOp(ConcatExpr, '<<' / '>>', ErrShiftExpr)
ConcatExpr    <- AddExpr ('..' ConcatExpr^ErrConcatExpr)?
AddExpr       <- @chainOp(MulExpr, '+' / '-', ErrAddExpr)
MulExpr       <- @chainOp(UnaryExpr, '*' / '//' / '/' / '%', ErrMulExpr)
UnaryExpr     <- ('not' / '-' / '#' / '~') UnaryExpr^ErrUnaryExpr
               / PowExpr
PowExpr       <- SimpleExp ('^' PowExpr^ErrPowExpr)?

SimpleExpr    <- Number / String / 'nil' / 'false' / 'true' / '...'
               / FuncDef / Table / SuffixedExpr

FuncCall      <- SuffixedExpr  -- only function or method calls are matched
VarExpr       <- SuffixedExpr  -- only identifiers or indexing are matched

SuffixedExpr <- PrimaryExpr (Index / Call)*
PrimaryExpr   <- Name / '(' Expr^ErrExprParen ')'^ErrCParenExpr
Index         <- '.' !'.' Name^ErrNameIndex / '[' !('=' / '[') Expr^ErrExprIndex ']'^ErrCBracketIndex
Call          <- ':' !':' Name^ErrNameMeth FuncArgs^ErrMethArgs / FuncArgs

FuncDef       <- 'function' FuncBody
FuncArgs      <- '(' @commaSep(Expr, ErrArgList) ')'^ErrCParenArgs / Table / String

Table         <- '{' FieldList? '}'^ErrCBraceTable
FieldList     <- @sepBy(Field, FieldSep) FieldSep?
Field         <- FieldKey '='^ErrEqField Expr^ErrExprField / Expr
FieldKey      <- '[' !('=' / '[') Expr^ErrExprFKey ']'^ErrCBracketFKey / Name &('=' !'=')
FieldSep      <- ',' / ';'


@kw(p)        <- @token(p !IdRest)
@sym(p)       <- @token(p)
@token(p)     <- p skip
```

```
Skip          <- (Space / Comment)*
Space         <- space+
Comment       <- '--' LongStr / '--' (!'\n' .)*

Name          <- @token(!Reserved Ident)
Reserved      <- Keywords !IdRest
Keywords      <- 'and' / 'break' / 'do' / 'elseif' / 'else' / 'end'
               / 'false' / 'for' / 'function' / 'goto' / 'if' / 'in' / 'local'
               / 'nil' / 'not' / 'or' / 'repeat' / 'return' / 'then' / 'true'
               / 'until' / 'while'
Ident         <- IdStart IdRest*
IdStart       <- alpha / '_'
IdRest        <- alnum / '_'

Number        <- @token(Hex / Float / Int)
Hex           <- ('0x' / '0X') xdigit+^ErrDigitHex
Float         <- Decimal Expo? / digit+ Expo
Decimal       <- digit+ '.' digit* / '.' !'.' digit+^ErrDigitDeci
Expo          <- [eE] [+-]? digit+^ErrDigitExpo
Int           <- digit+

String        <- @token(ShortStr / LongStr)
ShortStr      <- '"' (EscSeq / !('"' / '\n') .)* '"'^ErrQuote
               / '\'' (EscSeq / !('\' / '\n') .)* '\''^ErrQuote
EscSeq        <- '\\' ( 'a' / 'b' / 'f' / 'n' / 'r' / 't' / 'v'
                      / '\n' / '\r' / '\\' / '"' / '\'' / 'z' space*
                      / digit digit? Digit?
                      / 'x' (xdigit xdigit)^ErrHexEsc
                      / 'u' '{'^ErrOBraceUEsc xdigit+^ErrDigitUEsc '}'^ErrCBraceUEsc
                      / ErrEscSeq
                      )

LongStr       <- Open (!Close .)* Close^ErrCloseLStr
                 -- Close must have same # of '='s as Open
Open          <- '[' Equals '[' '\n'?
Close         <- ']' Equals ']'
Equals        <- '='*

alnum         <- <lpeg.alnum>   -- [A-Za-z0-9]
alpha         <- <lpeg.alpha>   -- [A-Za-z]
digit         <- <lpeg.digit>   -- [0-9]
space         <- <lpeg.space>   -- [ \t\n]
Xdigit        <- <lpeg.xdigit>  -- [0-9a-fA-F]
```

Rationale for the Changes Made

1. Introduction of `BlockEnd`
    - This allows invalid statements to be reported
    - This replaces the previous change that allowed labels to be thrown on blocks (previous errors detected will also be detected by this new rule, and the new rule is better for recovery too)

2. Change of `ExprStat` into `FuncCall` / `Assignment`
    - To better match the rules with the actual Lua grammar
    - Previously, some additional checks had to be done via match-time capture
    - Now, the checks are much cleaner and more straightforward
    - Unfortunately, the checks cannot be implemented without match-time captures due to the left recursive nature of `SuffixedExpr`

3. Additional checks when matching certain symbols ('~', '[', '.', '=')
    - Added to allow additional labels to be put

4. Short string newline and escape sequences
    - Fixed the rules on short strings to match the actual grammar of Lua 5.3

5. Splitting and renaming of rules, and other minor changes (like `@commaSep`)
    - Mostly done for clarity
    - Should not affect the matching except the change in the rule `Shebang`

Labels and Error Messages

1.	ErrExtra	- unexpected character(s), expected EOF
2.	ErrInvalidStat	- unexpected token, invalid start of statement
3.	ErrEndIf	- expected 'end' to close the if statement
4.	ErrExprIf	- expected a condition after 'if'
5.	ErrThenIf	- expected 'then' after the condition
6.	ErrExprEIf	- expected a condition after 'elseif'
7.	ErrThenEIf	- expected 'then' after the condition
8.	ErrEndDo	- expected 'end' to close the do block
9.	ErrExprWhile	- expected a condition after 'while'
10.	ErrDoWhile	- expected 'do' after the condition
11.	ErrEndWhile	- expected 'end' to close the while loop
12.	ErrUntilRep	- expected 'until' at the end of the repeat loop
13.	ErrExprRep	- expected a conditions after 'until'
14.	ErrForRange	- expected a numeric or generic range after 'for'
15.	ErrEndFor	- expected 'end' to close the for loop
16.	ErrExprFor1	- expected a starting expression for the numeric range
17.	ErrCommaFor	- expected ',' to split the start and end of the range
18.	ErrExprFor2	- expected an ending expression for the numeric range
19.	ErrExprFor3	- expected a step expression for the numeric range after ','
20.	ErrInFor	- expected '=' or 'in' after the variable(s)
21.	ErrEListFor	- expected one or more expressions after 'in'
22.	ErrDoFor	- expected 'do' after the range of the for loop
23.	ErrDefLocal	- expected a function definition or assignment after local
24.	ErrNameLFunc	- expected a function name after 'function'
25.	ErrEListLAssign	- expected one or more expressions after '='
26.	ErrEListAssign	- expected one or more expressions after '='
27.	ErrFuncName	- expected a function name after 'function'
28.	ErrNameFunc1	- expected a function name after '.'
29.	ErrNameFunc2	- expected a method name after ':'
30.	ErrOParenPList	- expected '(' for the parameter list
31.	ErrCParenPList	- expected ')' to close the parameter list
32.	ErrEndFunc	- expected 'end' to close the function body
33.	ErrParList	- expected a variable name or '...' after ','
34.	ErrLabel	- expected a label name after '::'
35.	ErrCloseLabel	- expected '::' after the label
36.	ErrGoto	- expected a label after 'goto'
37.	ErrRetList	- expected an expression after ',' in the return statement
38.	ErrVarList	- expected a variable name after ','

39. ErrExprList	- expected an expression after ','
40. ErrOrExpr	- expected an expression after 'or'
41. ErrAndExpr	- expected an expression after 'and'
42. ErrRelExpr	- expected an expression after the relational operator
43. ErrBOrExpr	- expected an expression after '|'
44. ErrBXorExpr	- expected an expression after '~'
45. ErrBAndExpr	- expected an expression after '&'
46. ErrShiftExpr	- expected an expression after the bit shift
47. ErrConcatExpr	- expected an expression after '..'
48. ErrAddExpr	- expected an expression after the additive operator
49. ErrMulExpr	- expected an expression after the multiplicative operator
50. ErrUnaryExpr	- expected an expression after the unary operator
51. ErrPowExpr	- expected an expression after '^'
52. ErrExprParen	- expected an expression after '('
53. ErrCParenExpr	- expected ')' to close the expression
54. ErrNameIndex	- expected a field name after '.'
55. ErrExprIndex	- expected an expression after '['
56. ErrCBracketIndex	- expected ']' to close the indexing expression
57. ErrNameMeth	- expected a method name after ':'
58. ErrMethArgs	- expected some arguments for the method call (or '()')
59. ErrArgList	- expected an expression after ',' in the argument list
60. ErrCParenArgs	- expected ')' to close the argument list
61. ErrCBraceTable	- expected '}' to close the table constructor (or you missed a ',' or ';')
62. ErrEqField	- expected '=' after the table key
63. ErrExprField	- expected an expression after '='
64. ErrExprFKey	- expected an expression after '[' for the table key
65. ErrCBracketFKey	- expected ']' to close the table key
66. ErrDigitHex	- expected one or more hexadecimal digits after '0x'
67. ErrDigitDeci	- expected one or more digits after the decimal point
68. ErrDigitExpo	- expected one or more digits for the exponent
69. ErrQuote	- unclosed string
70. ErrHexEsc	- expected exactly two hexadecimal digits after '\x'
71. ErrOBraceUEsc	- expected '{' after '\u'
72. ErrDigitUEsc	- expected one or more hexadecimal digits for the UTF-8 code point
73. ErrCBraceUEsc	- expected '}' after the code point
74. ErrEscSeq	- invalid escape sequence
75. ErrCloseLStr	- unclosed long string

Examples of Each Error

Errors are highlighted in yellow, but the actual error reported may be different and if it does, it is highlighted in teal or underlined if it overlaps. Note: an invisible '_' is used to highlight spaces.

1. **ErrExtra - unexpected character(s), expected EOF**

```
a. return; print("hello")
b. while foo do if bar then baz() end end end
c. local func f()
     g()
   end
d. function qux()
     if false then
       -- do
       return 0
     end
   end
   return 1
   end
   print(qux())
```

2. **ErrInvalidStat - unexpected token, invalid start of statement**

```
a. find_solution() ? print("yes") : print("no")
b. local i : int = 0
c. local a = 1, b = 2
d. x = -
   y = 2
e. obj::hello()
f. while foo() do
     // not a lua comment
     bar()
   end
g. repeat:
     action()
   until condition
h. function f(x)
     local result
     ... -- TODO: compute for the next result
     return result
   end
i. x;
j. a, b, c
k. local x = 42 // the meaning of life
l. let x = 2
m. if p then
     f()
   elif q then
     g()
   end
n. function foo()
     bar()
   emd
```

3. ErrEndIf **- expected 'end' to close the if statement**

```
a. if 1 > 2 then print("impossible")  
b. if 1 > 2 then return; print("impossible") end
c. if condA then doThis()
   else if condB then doThat() end  
d. if a then
     b()
   else
     c()
   else
     d()
   end
```

4. ErrExprIf **- expected a condition after 'if'**

```
a. if   then print("that") end
b. if !ok then error("fail") end
```

5. ErrThenIf **- expected 'then' after the condition**

```
a. if age < 18  
     print("too young!")
   end
```

6. ErrExprElf **- expected a condition after 'elseif'**

```
a. if age < 18 then print("too young!")
   elseif   then print("too old") end
```

7. ErrThenElf **- expected 'then' after the condition**

```
a. if not result then error("fail")
   elseif result > 0:
     process(result)
   end
```

8. ErrEndDo **- expected 'end' to close the do block**

```
a. do something()  
b. do
     return arr[i]
     i = i + 1
   end
```

9. ErrExprWhile **- expected a condition after 'while'**

```
a. while !done do done = work() end
b. while   do print("hello again!") end
```

10. ErrDoWhile **- expected 'do' after the condition**

```
a. while not done then work() end
b. while not done  
     work()
   end
```

11. ErrEndWhile **- expected 'end' to close the while loop**

```
a. while not found do i = i + 1
b. while i < #arr do
     if arr[i] == target then break
     i = i +1
   end
```

12. ErrUntilRep **- expected 'until' at the end of the repeat loop**

```
a. repeat play_song()
```

13. ErrExprRep **- expected a conditions after 'until'**

```
a. repeat film() until end
```

14. ErrForRange **- expected a numeric or generic range after 'for'**

```
a. for (key, val) in obj do
     print(key .. " -> " .. val)
   end
```

15. ErrEndFor **- expected 'end' to close the for loop**

```
a. for i = 1,10 do print(i)
```

16. ErrExprFor1 **- expected a starting expression for the numeric range**

```
a. for i =  ,10 do print(i) end
```

17. ErrCommaFor **- expected ',' to split the start and end of the range**

```
a. for i = 1 to 10 do print(i) end
```

18. ErrExprFor2 **- expected an ending expression for the numeric range**

```
a. for i = 1,  do print(i) end
```

19. ErrExprFor3 **- expected a step expression for the numeric range after ','**

```
a. for i = 1,10,  do print(i) end
```

20. ErrInFor **- expected '=' or 'in' after the variable(s)**

```
a. for arr do print(arr[i]) end
b. for nums := 1,10 do print(i) end
```

21. ErrEListFor **- expected one or more expressions after 'in'**

a. for i in ? do print(i) end

22. ErrDoFor **- expected 'do' after the range of the for loop**

a. for i = 1,10 doo print(i) end
b. for _, elem in ipairs(list)
   print(elem)
   end

23. ErrDefLocal **- expected a function definition or assignment after local**

```
a. local
b. local; x = 2
c. local *p = nil
```

24. ErrNameLFunc **- expected a function name after 'function'**

```
a. local function() return 0 end
b. local function 3dprint(x, y, z) end
c. local function repeat(f, ntimes) for i = 1,ntimes do f() end end
```

25. ErrEListLAssign **- expected one or more expressions after '='**

```
a. local x = ?
```

26. ErrEListAssign **- expected one or more expressions after '='**

```
a. x = ?
```

27. ErrFuncName - expected a function name after 'function'

```
a. function() return 0 end
b. function 3dprint(x, y, z) end
c. function repeat(f, ntimes) for i = 1,ntimes do f() end end
```

28. ErrNameFunc1 - expected a function name after '.'

```
a. function foo.() end
b. function foo.1() end
```

29. ErrNameFunc2 - expected a method name after ':'

```
a. function foo:() end
b. function foo:1() end
```

30. ErrOParenPList - expected '(' for the parameter list

```
a. function foo
     return bar
   end
b. function foo?(bar)
     return bar
   end
```

31. ErrCParenPList - expected ')' to close the parameter list

```
a. function foo(bar
     return bar
   end
b. function foo(bar; baz)
     return bar
   end
c. function foo(a, b, ...rest) end
```

32. ErrEndFunc - expected 'end' to close the function body

```
a. function foo(bar)
     return bar

b. function foo() do
     bar()
   end
```

33. ErrParList - expected a variable name or '...' after ','

```
a. function foo(bar, baz,)
     return bar
   end
```

34. ErrLabel - expected a label name after '::'

```
a. ::1::
```

35. ErrCloseLabel - expected '::' after the label

```
a. ::loop
```

36. ErrGoto - expected a label after 'goto'

```
a. goto;
b. goto 1
```

37. ErrRetList - expected an expression after ',' in the return statement

```
a. return a, b,
```

38. ErrVarList - expected a variable name after ','

```
a. x, y, = 0, 0
```
39. **ErrExprList** - **expected an expression after ','**
```
a. x, y = 0, 0,
```

40. **ErrOrExpr** - **expected an expression after 'or'**
```
a. foo(a or  )
b. x = a or $b
```
41. **ErrAndExpr** - **expected an expression after 'and'**
```
a. foo(a and  )
b. x = a and $b
```
42. **ErrRelExpr** - **expected an expression after the relational operator**
```
a. foo(a <  )
b. x = a < $b
c. foo(a <=  )
d. x = a <= $b
e. foo(a >  )
f. x = a > $b
g. foo(a >=  )
h. x = a >= $b
i. foo(a ==  )
j. x = a == $b
k. foo(a ~=  )
l. x = a ~= $b
```
43. **ErrBOrExpr** - **expected an expression after '|'**
```
a. foo(a |  )
b. x = a | $b
```
44. **ErrBXorExpr** - **expected an expression after '~'**
```
a. foo(a ~  )
b. x = a ~ $b
```
45. **ErrBAndExpr** - **expected an expression after '&'**
```
a. foo(a &  )
b. x = a & $b
```
46. **ErrShiftExpr** - **expected an expression after the bit shift**
```
a. foo(a >>  )
b. x = a >> $b
c. foo(a <<  )
d. x = a >> $b
e. x = a >>> b
```
47. **ErrConcatExpr** - **expected an expression after '..'**
```
a. foo(a ..  )
b. x = a .. $b
```
48. **ErrAddExpr** - **expected an expression after the additive operator**
```
a. foo(a +  , b)
b. x = a + $b
c. foo(a -  , b)
d. x = a - $b
e. arr[i++]
```
49. **ErrMulExpr** - **expected an expression after the multiplicative operator**

```
a. foo(b, a * )
b. x = a * $b
c. foo(b, a / )
d. x = a / $b
e. foo(b, a // )
f. x = a // $b
g. foo(b, a % )
h. x = a % $b
```

50. ErrUnaryExpr - expected an expression after the unary operator

```
a. x, y = a + not , b
b. x, y = a + - , b
c. x, y = a + # , b
d. x, y = a + ~ , b
```

51. ErrPowExpr - expected an expression after '^'

```
a. foo(a ^ )
b. x = a ^ $b
```

52. ErrExprParen - expected an expression after '('

```
a. x = ( )
b. y = (???)
```

53. ErrCParenExpr - expected ')' to close the expression

```
a. z = a*(b+c
b. w = (0xBV)
c. ans = 2^(m*(n-1)
```

54. ErrNameIndex - expected a field name after '.'

```
a. f = t.
b. f = t.['f']
c. x.
```

55. ErrExprIndex - expected an expression after '['

```
a. f = t[ ]
b. f = t[?]
```

56. ErrCBracketIndex- expected ']' to close the indexing expression

```
a. f = t[x[y]
b. f = t[x,y]
c. arr[i--]
```

57. ErrNameMeth - expected a method name after ':'

```
a. x = obj:
b. x := 0
```

58. ErrMethArgs - expected some arguments for the method call (or '()')

```
a. cow:moo
b. dog:bark msg
c. duck:quack[4]
d. local t = {
     x = X:
     y = Y;
   }
```

59. ErrArgList - expected an expression after ',' in the argument list

```
a. foo(a, b, )
b. foo(a, b, ..)
```

60. ErrCParenArgs - expected ')' to close the argument list

```
a. foo(a + (b - c)
b. foo(arg1 arg2)
```

61. ErrCBraceTable - expected '}' to close the table constructor (or you missed a ',' or ';')

```
a. nums = {1, 2, 3]
b. t = {
     one = 1;
     two = 2
     three = 3;
     four = 4
   }
```

62. ErrEqField - expected '=' after the table key

```
a. words2nums = { ['one'] -> 1 }
```

63. ErrExprField - expected an expression after '='

```
a. words2nums = { ['one'] => 2 }
```

64. ErrExprFKey - expected an expression after '[' for the table key

```
a. table = { [ ] = value }
```

65. ErrCBracketFKey - expected ']' to close the table key

```
a. table = { [key  = value }
```

66. ErrDigitHex - expected one or more hexadecimal digits after '0x'

```
a. print(0x )
b. print(0xGG)
```

67. ErrDigitDeci - expected one or more digits after the decimal point

```
a. print(1 + . 0625)
b. print(. )
```

68. ErrDigitExpo - expected one or more digits for the exponent

```
a. print(1.0E )
b. print(3E )
```

69. ErrQuote - unclosed string

```
a. local message = "Hello
b. local message = "*******
   Welcome
   *******"
c. local message = 'Hello
d. local message = '*******
   Welcome
   *******'
```

70. ErrHexEsc - expected exactly two hexadecimal digits after '\x'

```
a. print("\x ")
b. print("\xF ")
c. print("\xG")
```

71. ErrOBraceUEsc - expected '{' after '\u'

```
a. print("\u 3D")
```

72. **ErrDigitUEsc** **- expected one or more hexadecimal digits for the UTF-8 code point**

   a. `print("\u{ }")`
   b. `print("\u{XD}")`

73. **ErrCBraceUEsc** **- expected '}' after the code point**

   a. `print("\u{0x3D}")`
   b. `print("\u{FFFF Hi")`

74. **ErrEscSeq** **- invalid escape sequence**

   a. print("\m")

75. **ErrCloseLStr** **- unclosed long string**

   a.
```
local message = [==[
*******
WELCOME
*******
]= ]
```

Error Reporting and Error Recovery

Error Reporting

Error reporting no longer gives a list of expected tokens. Instead, it gives a more specific error message (as can be seen in the examples in the previous section). The line and column number of the error is still reported, but though the line itself is not printed.

Recovery Strategies

The ideas behind the design of the recovery strategies are as follows:

- simple statements (i.e. statements without nesting) often take exactly one line
- it is unlikely for there to be more than one error in a simple statement
- a common exception to the single line rule are statements with table constructors
- strategies should be fairly to conservative to avoid introducing false and cascading errors

With these ideas in mind, the following strategies were employed:

1. For simple statements, when an error occurs within the statement (usually bubbled up from an expression), skip the line (which hopefully contains the full statement)
    - by skipping a line, we are likely to resume at the start of the next statement, which also helps prevents introducing cascading of errors
    - by skipping a line, we automatically remove the possibility of catching other errors within the same line, but this should be fine as it is an unlikely situation
    - some additional rules might also be skipped for some statements (example: when there's an error in the if condition, it will skip the line and the rule checking for the "then" keyword that should follow as it is most likely on the same line that was skipped)
    - to better handle multiple statements on a single line, the parser may skip to the first semicolon or "end" on the same line instead
    - to handle tables better, the parser may also skip to the first semicolon on the same line or comma if it is at the end of the same line
2. For other errors, the parser records the error and resumes at the point of error, as if there was no error, sometimes capturing a dummy
    - although imperfect, it is the best we can do since it is impossible to surely know the best point to resume parsing
    - skipping a line would be impractical as it would most likely skip important tokens on the same line (like how the name of a function is on the same line as its parameters)
    - this is the usual strategy used for most keywords (e.g. missing "then")

To prevent overwhelming the user with errors (especially if they were wrongly introduced due to the recovery strategy), the parser has a limit of a maximum of 20 errors reported. Validation of the AST (like checking of goto labels) is only done if no syntax errors are found.

Comparison of Error Reporting Before and After

In general, the error reporting of the labeled parser is more direct and therefore clearer. It usually provides better context compared to the generic error messages given by the original parser.

Take for example the following code snippet:

```
for i=1,10, do end
```

In the original parser, the following error is reported:

```
test.lua:1:13: syntax error, unexpected 'do', expecting '(', 'Name', '{', 'function', '...', 'true', 'false', 'nil', 'String', 'Number', '~', '#', '-', 'not'
```

While in the labeled parser, the following error is reported:

```
test.lua:1:13: syntax error, expected a step expression for the numeric range after ','
```

In the original parser, the unexpected token is shown. And while it is indeed unexpected, it is a bit confusing because the do token itself is not the error, but the missing expression before it. Another thing one would notice is the long list of tokens shown. Due to the length of the list, not only is it unhelpful, it is also intimidating! A much clearer and concise way to express this would be to simply say that an expression is expected.

In the labeled parser, the flaws of the original parser are absent. Not only is it mentioned that an expression is expected, it also mentioned that the expression is supposed to be the step value of the for loop's numeric range. Additionally, a hint was mentioned at the end to inform the user that a comma triggered the expectation.

Another similar example is this snippet:

```
if a:any then else end
```

In the original parser, the following error is reported:

```
test.lua:1:10: syntax error, unexpected 'then', expecting 'String', '{', '('
```

While in the labeled parser, the following error is reported:

```
test.lua:1:10: syntax error, expected some arguments for the method call (or '()')
```

For this example, in the labeled parser, it is immediately clear for the programmer that arguments for a method call were expected. However, in the original parser, it is not obvious why 'String', '{', '(' are expected.

Of course, the price for having better error messages is the additional manual labor done writing down the error messages and putting of the labels (although this can be done semi-automatically). On the other hand, the original error detection and reporting mechanism was a fully automatic process.

There are certain cases that the original parser provides slightly better error messages. For example:

```
for k;v in pairs(t) do end
```

In the original parser, the following error is reported:

```
test.lua:1:6: syntax error, unexpected ';', expecting 'in', ',', '='
```

While in the labeled parser, the following error is reported:

```
test.lua:1:6: syntax error, expected '=' or 'in' after the variable(s)
```

Although both parsers detected the same error, the labeled parser gives a more narrow set of choices. The original parser has a ',' in the list of expected tokens that could allow the programmer to realize the commas are used to separate the variables and not semicolons. For more experienced programmers, this shouldn't be a big issue though as they would know what is correct.

Unfortunately, in some cases, both parsers still give poor error messages. For example:

```
local function test(...,a) end
```

In the original parser, the following error is reported:

```
test.lua:1:24: syntax error, unexpected ',', expecting ')'
```

While in the labeled parser, the following error is reported:

```
test.lua:1:24: syntax error, expected ')' to close the parameter list
```

In this example, both parsers give similar error messages. Again, the labeled parser provides more context by specifying that the ')' is needed for closing the parameter list. Despite this, both of them are still confusing since a ')' does exist to close the parameter list later on. The real error here is not using '...' as the final argument. One way to improve this specific example would be to put checking in a different function and not in the grammar itself. Similarly, some other rules (like having statements after a return) can be made more lenient only to be checked later to improve the error reporting.

Benchmarks

Benchmarks were run to measure the performance impact of introducing labels (without error recovery). Surprisingly, the labeled version performed significantly better than the original. Lua's `os.clock` was used for measuring the time. Tests were run on an Intel Core 2 Duo (2.33GHz) desktop with Windows 8.1 and Lua v5.2.4 installed.

Below are the average times taken to completely parse all repositories (5 runs each):

- original: 42.429s
- labeled: 2.5636s

On average, the labeled version was **16.55x faster** than the original version. Checks were also done to ensure that both version produced the same AST. Except for the differences in the handling of escape codes*****, the ASTs were identical.

The cause of the massive difference in speed is due to the overhead of the original error detection and reporting system of recording the farthest failure position (ffp) and list of expected tokens at that point. Even when the input is completely valid, the system will still have to record all failures in the ordered choices. Removing the old system shortened the parse time by ~5.5x.

Another cause of the speed difference is an inefficient implementation of parsing two statements: function calls and assignment statements without "local". Fixing these rules furthered shortened the parse time by ~2.5x. At this point, the original is already quite close to the labeled version but still slower by a second. This difference is probably due to the other refactoring work done on the code.

***** Aside from handling more escape codes, the labeled version also fixes a bug in the original where strings like "\\n" would be parsed as a newline (replacing "\n" first and then replacing "\<newline>"), instead of the expected "\n" (only replacing "\\").

The following repositories were included in the benchmarks:

- **Eve** - an experimental programming platform (https://github.com/witheve/Eve)
- **Kong**** - an API Gateway built on NGINX (https://github.com/Mashape/kong)
- **KOReader** - an ebook reader (https://github.com/koreader/koreader)
- **LuaCov** - coverage analyzer for Lua (https://github.com/keplerproject/luacov)
- **LuaRocks** - the Lua package manager (https://github.com/keplerproject/luarocks)
- **Mr. Rescue** - an acade-style fire fighting game (https://github.com/SimonLarsen/mrrescue)
- **Sailor** - a Lua MVC Web Framework (https://github.com/sailorproject/sailor)

These repositories were chosen based on popularity, diversity, and heavy use of (pure) Lua.

****** Kong was originally included in the benchmarks but later removed in the average as it skewed results too much in favor of the labeled parser.

Limitations and Other Possible Improvements

- Currently, due to how the tokens are parsed, the error position reported can sometimes be confusing (as it skips whitespace including comments). Additional changes can be made to fix this at the cost of a slight increase in code complexity. (This requires modifying the token function to keep track of the last pre-whitespace position encountered).
- A preview of the erroneous line with a caret pointing to the error can be printed as well, but this will only be helpful if the error positions are properly adjusted (see the previous bullet point). This can be implemented similar to how it was done for LPegLabel's relabel parser.
- For some errors, it may be helpful to point to or indicate the start of the error (for example long strings and unclosed statements -- like how lua does it).
- Some error messages can be made more specific. For example, instead of saying "expected an expression after the unary operator", the specific unary operator should be mentioned instead. One way to do this would be to split each operator to a different case, but doing so would make the grammar needlessly more complex. A better way to do this would be to inspect the captures at the point of error, but this is currently not possible.
- Some rules in the grammar can be made more lenient to improve error reporting. For example, allowing statements after a return (and then reporting them later) would allow the parser to continue checking those statements too and also give a clearer error message.
- Sometimes error messages can be confusing. For example, it may report a missing closing brace "}" in the table constructor when the true error is a missing comma "," or semicolon ";". However, it is difficult to come up with an algorithm to generate better error messages as it will most likely require some lookahead and heavy use of heuristics (which will definitely be much more complex as well). Clinton Jeffery's paper: "Generating LR Syntax Error Messages from Examples," presents an interesting approach in attempting to solve this problem in LR parsers.
- The recovery mechanism has not been tested on real world usage, and so the effectiveness of its heuristics remains to be seen. It would be good to do some real world testing to better understand the nature of syntax errors in practice.
- Of course, the recovery mechanism is imperfect and may introduce false errors. This is definitely a hard problem to solve and still haunts most compilers until today (for example, try mistyping "for" as "fro" in C or Java).