Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт прикладной математики и механики
Высшая школа кибербезопасности и защиты информации

КУРСОВАЯ РАБОТА

«Создание распознавателя языка Lua с помощью YACC и Lex»

по дисциплине «Формальные грамматики и теория компиляторов»

Выполнил
студент гр. 3651003/70801 Гасанов Э.А.

<подпись>

Преподаватель
старший преподаватель Семьянов П.В.

<подпись>

Санкт-Петербург
2020

СОДЕРЖАНИЕ

Введение

Целью данной курсовой работы является проверить возможность создания грамматики для современного языка с помощью YACC и Lex. А также проверить, насколько YACC и Lex соответствуют требованиям, которые предъявляют современные языки. Действительно ли с помощью YACC и Lex удастся создать парсер для современных языков. Эти вопросы актуальны по сей день, потому что каждый, кто хочет разобраться как работают компиляторы, должен понять, какова роль инструментов YACC и Lex в разработке компиляторов. Для того чтобы ответить на вышеупомянутые вопросы, мы создадим парсер Си-подобного языка Lua и опишем процесс создания грамматики этого языка, то есть методикой данного исследования является создание неидеального парсера.

Теперь необходимо перечислить задачи, необходимые как для создания парсера языка Lua, так и для достижения поставленной цели:

1. Создать лексический анализатор для токенизации.
2. Сформулировать формальные правила для основных видов инструкций языка Lua: блоки, таблицы, функции и так далее.
3. Разрешить возникающие конфликты грамматики, но если останутся тяжело разрешаемые конфликты – оставить их неразрешенными, так как мы не пишем идеальный парсер.
4. Лексер должен распознавать все возможные виды вещественных чисел и комментариев, поддерживаемых Lua.

1. Лексика языка Lua

1.1. Токены

Разбор языка начинается с лексического анализатора. С помощью лексера Lex были выделены основные токены:

Токен	Регулярное выражение	Описание
DO	do	Ключевые слова
WHILE	while	
FOR	for	
UNTIL	until	
REPEAT	repeat	
END	end	
IN	in	
IF	if	
THEN	then	
ELSEIF	elseif	
ELSE	else	
LOCAL	local	
FUNCTION	function	
RETURN	return	
BREAK	break	
GOTO	goto	
NOT	not	
NIL	nil	
FALSE	false	
TRUE	true	
AND	and	
OR	or	
PLUS	[+]	

MINUS	[-]	Арифметические операции
TIMES	[*]	
DIVIDE	[/]	
POWER	[\^]	
MODULO	[%]	
ASSIGN	[=]	
TILDE	[~]	Логические операции
PIPE	[|]	
AMPERSAND	[&]	
LEFT_SHIFT	[<][<]	Операции сравнения
RIGHT_SHIFT	[>][>]	
EQUALS	[=][=]	
LESS_THAN	[<]	
LESS_EQUALS_THAN	[<][=]	
MORE_THAN	[>]	
MORE_EQUALS_THAN	[>][=]	
APPEND	[.][.]	Конкатенация строк
SQUARE	[#]	Размер объекта
STRING	\"([^\\\"]|\\\n|\\.)*\" \'([^\\\']|\\\n|\\.)*\' Функция parseBlockString	
NAME	[A-Za-z_][A-Za-z0-9_]*	Имена переменных или функций
LABEL	::[A-Za-z_][A-Za-z0-9_]*::	Метки
DOT	\.	Разделители

COLON	:	
COMMA	,	
SEMICOLON	;	
PARANTHESES_L	\(	
PARANTHESES_R	\)	
BRACES_L	\{	
BRACES_R	\}	
BRACKET_L	\[	
BRACKET_R	\]	

Таблица 1, лексика языка Lua.

1.2. Ключевые слова

Следующие ключевые слова (keywords) зарезервированы и не могут использоваться в качестве имен:

do	Начало блока
end	Конец блока
for	Операторы цикла
while	
repeat	
until	
if	Условные операторы
then	
elseif	
goto	Оператор безусловного перехода
true	Логические константы
false	
or	Логические операторы

and	
not	
nil	Нулевое значение
function	Объявление функции
return	Возврат из функции
local	Спецификатор области видимости

Таблица 2, ключевые слова Lua.

Lua является языком чувствительным к регистру: and — это зарезервированное слово, но And и AND это два разных, допустимых имени. По соглашению, программы должны избегать создания имен, начинающихся с символа подчеркивания и последующим за ним одной или более букв верхнего регистра (как например, _VERSION).[1]

2. Создание грамматики

Для разработки грамматики были выделены основные принципы построения программы языка Lua:

- Программы состоит из блоков
- Каждый блок состоит из некоторых инструкций
- Инструкции могут быть как выражениями, так и целыми блоками

2.1. Блоки

Каждый блок программы можно описать как некоторую сущность, который состоит из своего тела и конечного состояния.

```
unit : piece_code
;
piece_code : futher_piece_code last_piece_code
| futher_piece_code
| last_piece_code
;
```

Листинг 1, формальное описание блока.

Некоторые из инструкций могут оказаться return или break, поэтому их тоже необходимо формально описать:

```
last_piece_code: RETURN datagroup
| RETURN
| BREAK
;
```

Листинг 2, описание конечных состояний блока: цикла или функции.

Тело блока выражено нетерминалом instruction и его составляющими:

```
instruction   : seqgroup ASSIGN datagroup
| LOCAL namegroup ASSIGN datagroup
| LOCAL namegroup
| FUNCTION name_routine structure_routine
| LOCAL FUNCTION NAME structure_routine
| frontdata
| DO unit END
| DO END
| whileunit
| REPEAT unit UNTIL data
| REPEAT UNTIL data
| ifunit
| forunit
| GOTO NAME
| LABEL
| SEMICOLON
| MYEOF {return 0;}
;
```

Листинг 3, описание ключевых инструкций языка Lua.

2.2. Инструкции

Инструкции – это основные конструкции языка, которые определяют логику работы блока. Рассмотрим построения основных инструкций на примере конструкции цикла «for»:

```
forunit: FOR NAME ASSIGN data COMMA data DO unit END
| FOR NAME ASSIGN data COMMA data COMMA data DO unit END
| FOR namegroup IN datagroup DO unit END
| FOR NAME ASSIGN data COMMA data DO END
| FOR NAME ASSIGN data COMMA data COMMA data DO END
| FOR namegroup IN datagroup DO END
;
```

Листинг 4, формальное описание цикла for.

Отсюда видно, что данная конструкция состоит из токена FOR, который соответствует ключевому слову for, названию переменной, которой будет присвоено некоторое значение, выражений и блока «do».

Выражения могут принимать значения констант, переменных, разнообразных бинарных и унарных операций, массивов, таблиц, а также вызовов функций:

```
data: NIL
| FALSE
| TRUE
| NUMBER
| STRING
| TDOT
| routine
```

```
| frontdata
| hash_table
| data OR data
| data AND data
| data LESS_THAN data
| data LESS_EQUAL_THAN data
| data MORE_THAN data
| data MORE_EQUAL_THAN data
| data TILDE_EQUAL data
| data EQUALS data
| data APPEND data
| data PLUS data
| data MINUS data
| data TIMES data
| data DIVIDE data
| data MODULO data
| NOT data
| SQUARE data
| MINUS data
| data POWER data
| data LEFT_SHIFT data
| data RIGHT_SHIFT data
| data PIPE data
| data AMPERSAND data
| data TILDE data
| TILDE data
;
```

Листинг 5, выражения языка Lua.

2.3. Функции

Функции состоят из ключевого слова «function», параметров и тела функции:

```
routine: FUNCTION structure_routine
;
structure_routine: PARANTHESES_L parameters PARANTHESES_R unit END
| PARANTHESES_L PARANTHESES_R unit END
| PARANTHESES_L PARANTHESES_R END
| PARANTHESES_L parameters PARANTHESES_R END
;
```

Листинг 6, формальное представление функций.

Параметры функции принимают значения названия переменных, записанных через запятую:

```
parameters   : namegroup
| namegroup COMMA TDOT
| TDOT
;
```

Листинг 7, параметры функции.

2.4. Таблицы

Таблицы – это динамическая структура данных(объект), реализованная по принципу ассоциативных массивов. Ассоциативный массив — это массив, который можно индексировать не только с помощью чисел, но и со строками или любым другим значением языка, кроме nil. Из-за использования сборщика мусора, мы можем динамически добавлять в таблицу столько элементов, сколько захотим. Таблицы являются единственным механизмом структурирования данных в Lua. Таблицы могут использоваться для представления массивов, матриц, стеков, словарей, очередей и других структур данных. Для того чтобы создать простейшую таблицу, воспользуемся конструктором, которое в простейшей форме записывается как {}. Приступим разрабатывать грамматику таблиц:

```
hash_table: BRACES_L spotgroup BRACES_R
| BRACES_L BRACES_R ;
spot: BRACKET_L data BRACKET_R ASSIGN data
| NAME ASSIGN data
| data ;
spotgroup: next_spotgroup nextspot
;
next_spotgroup: spot | next_spotgroup spotdelimiter spot
nextspot: spotdelimiter  |  /* empty */
;
spotdelimiter: COMMA | SEMICOLON
;
```

Листинг 8, грамматика таблиц и конструктора таблиц в Lua.

Нетерминал hash_table описывает общее положение записанного конструктора таблицы: внутри терминальных левых и правых фигурных скобок (BRACES_L и BRACES_R соответственно) может как располагаться информация об элементах таблицы, так и вовсе не быть её. В первом случае происходит инициализация таблицы, а во втором – объявление таблицы как объекта. Если был выбран первый вариант – рекурсия продолжает углубляться. Нетерминал spotgroup следит за количеством записываемых объектов в таблицу. Например, в такую таблицу polyline={color=1} записывается только один элемент color. Тогда из нетерминала spotgroup вызовется нетерминал next_spotgroup, где вызывается только нетерминал spot(потому что записывается только один элемент color), в котором по шаблону выбираются подходящие терминалы, в данном случае это имя элемента таблицы(NAME) равенство (ASSIGN) и некая полезная нагрузка, которую опишем в следующем разделе. Затем возвращаемся на предыдущий уровень рекурсии в spotgroup, и затем в nextspot выберется пустой вариант, так как после color нет запятой, поэтому не выбирается spotdelimeter. Затем полученные результаты возвращаются в hash_table в фигурные скобки.

Если в таблицу записывается два или больше элементов, например, polyline={color=3, tall=4}, тогда мы попадаем из hash_table в spotgroup, где вызывается nextspotgroup, в котором вызовется spot с нужными терминалами. Затем возвращаясь на предыдущий уровень рекурсии, мы попадаем в next_spotgroup, где сохранено предыдущая(ие) обработка(и) элементов, перечисленных через запятую или точку с запятой. Затем проверяется знак-разделитель в spotdelimiter, и возвращаясь обратно в next_spotgroup вызывается spot для одного из следующих элементов, находящегося внутри фигурных скобок. Затем, когда все элементы внутри скобок обработаны и рекурсия выходит из next_spotgroup как выполненный нетерминал, вызывается nextspot, который должен определить, есть ли запятая после последнего элемента в фигурных скобках. Это необходимо (и на следующем

примере продемонстрируем), так как по правилам грамматики языка Lua, после элемента может стоять необязательная запятая. Но в этом случае выбирается empty и выполненный spotgroup возвращается в hash_table.

Теперь продемонстрируем, почему мы создали эту часть грамматики:

```
nextspot: spotdelimiter
| /* empty */
;
```

Листинг 9, грамматика, обрабатывающая последнюю не обязательную запятую.

Например, polyline= {color=3, tall=4,} имеет запятую после последнего элемента. Рекурсия аналогична предыдущим примерам, однако по возвращении в spotgroup, в nextspot выберется spotdelimiter, то есть либо терминал-запятая, либо терминал-точка с запятой. Затем рекурсия вернется в spot и продолжит выполнение.

2.5. Конфликты, которые удалось разрешить во время создания грамматики

В ходе работы было выявлено большое количество конфликтов при создании грамматики:

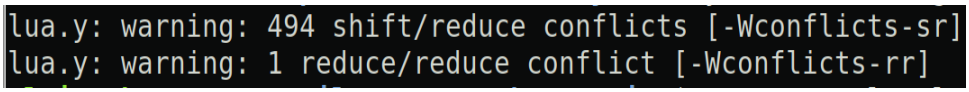

Рисунок 1, множество ошибок.

Эти конфликты означают, что грамматика, поданная YACC или Bison, не является LALR (1), поэтому они не могут решить, как корректно поступить при парсинге в той или иной конфликтной ситуации. Обычно YACC или Bison в конфликте shift/reduce выбирает shift, если иное не указано явным правилом, например %left или декларацией приоритетов операторов.

Запустимся в режиме отладки, выставив yydebug=1 в YACC-файле, чтобы создать файл y.output. В самом файла заметим перечень состояний:

State 10 conflicts: 3 shift/reduce
State 27 conflicts: 1 reduce/reduce
State 44 conflicts: 4 shift/reduce
State 78 conflicts: 20 shift/reduce
State 79 conflicts: 20 shift/reduce
State 80 conflicts: 20 shift/reduce
State 84 conflicts: 2 shift/reduce
State 88 conflicts: 20 shift/reduce
State 142 conflicts: 20 shift/reduce
State 143 conflicts: 20 shift/reduce
State 144 conflicts: 20 shift/reduce
State 145 conflicts: 20 shift/reduce

```
State 146 conflicts: 20 shift/reduce
State 147 conflicts: 20 shift/reduce
State 148 conflicts: 20 shift/reduce
State 149 conflicts: 20 shift/reduce
State 150 conflicts: 20 shift/reduce
State 151 conflicts: 20 shift/reduce
State 152 conflicts: 20 shift/reduce
State 153 conflicts: 20 shift/reduce
State 154 conflicts: 20 shift/reduce
State 155 conflicts: 20 shift/reduce
State 156 conflicts: 20 shift/reduce
State 157 conflicts: 20 shift/reduce
State 158 conflicts: 20 shift/reduce
State 159 conflicts: 20 shift/reduce
State 160 conflicts: 20 shift/reduce
State 161 conflicts: 20 shift/reduce
State 179 conflicts: 2 shift/reduce
State 189 conflicts: 1 shift/reduce
State 202 conflicts: 2 shift/reduce
```

Листинг 10, перечень состояний конфликтов.

Мы замечаем, что от номера 142 до 161 однотипные и относятся к одному правилу. Конфликтным оказалось следующее положение грамматики:

```
data: NIL
| FALSE
| TRUE
| NUMBER
```

| STRING

| TDOT

| routine

| frontdata

| hash_table

| data OR data

| data AND data

| data LESS_THAN data

| data LESS_EQUAL_THAN data

| data MORE_THAN data

| data MORE_EQUAL_THAN data

| data TILDE_EQUAL data

| data EQUALS data

| data APPEND data

| data PLUS data

| data MINUS data

| data TIMES data

| data DIVIDE data

| data MODULO data

| NOT data

| SQUARE data

| MINUS data

| data POWER data

| data LEFT_SHIFT data

| data RIGHT_SHIFT data

| data PIPE data

| data AMPERSAND data

| data TILDE data

| TILDE data

```
;
```

Листинг 11, синтаксис бинарных, унарных операций и выражений.

Ключевой ошибкой стал тот факт, что в этих правилах не понятно, является ли правило левоассоциативной или правоассоциативной операцией относительно токена. Ввод правила, определяющего данные операции левоассоциативными, позволило существенно сократить количество конфликтов. Поэтому в грамматике появилось следующее дополнение:

```
%left MINUS OR PLUS NOT TIMES DIVIDE POWER MODULO EQUALS LESS_THAN MORE_THAN LESS_EQUAL_THAN MORE_EQUAL_THAN TILDE_EQUAL AND SQUARE APPEND TILDE PIPE AMPERSAND LEFT_SHIFT RIGHT_SHIFT
```

Листинг 12, явное указание, что операции левоассоциативные.

После этого исправления количество конфликтов уменьшилось на 480 ошибок:

```
lua.y: warning: 14 shift/reduce conflicts [-Wconflicts-sr]
lua.y: warning: 1 reduce/reduce conflict [-Wconflicts-rr]
```

Рисунок 2, уменьшение количество ошибок, по сравнению с предыдущим шагом.

Продолжим отладку грамматики путём изучения файла y.output:

```
State 10 conflicts: 3 shift/reduce
State 27 conflicts: 1 reduce/reduce
State 44 conflicts: 4 shift/reduce
State 84 conflicts: 2 shift/reduce
State 179 conflicts: 2 shift/reduce
State 189 conflicts: 1 shift/reduce
State 202 conflicts: 2 shift/reduce
```

Листинг 13, оставшиеся конфликты.

Состояние 189 имеет одну ошибку типа shift/reduce, возникшей при описании синтаксиса ветвлений языка Lua.

Конфликтной оказалась эта часть грамматики:

```
ifunit: IF data THEN unit else {printf("if is here\n");}
;
else: END
| ELSE unit END
| elseifs
;
elseifs: elseifs ELSEIF data THEN unit else
| ELSEIF data THEN unit else
;
```

Листинг 14, неудачная грамматика ветвления.

Конфликт возникает, так как грамматика неоднозначна: может выполниться shift на parser stack, либо reduce из-за положения правила else: elseifs.

```
State 189
    else: elseifs . (rule 38)
    elseifs: elseifs . ELSEIF data THEN unit else (rule 39)

    ELSEIF  shift, and go to state 200

    ELSEIF    [reduce using rule 38 (else)]
    $default  reduce using rule 38 (else)
```

Листинг 15, shift/reduce конфликт состояния 189, отображенный в y.output.

Здесь мы можем видеть неоднозначность, решаемую YACC в пользу shift. То есть на parse stack сверху добавится терминал ELSEIF.

Чтобы решить описанную выше проблему, изменим грамматику ветвления на однозначную. Сменим подход в проектировании грамматики, добавив больше нетерминалов, то есть обозначим каждое ключевое слово ветвления(if,else,elseif) своим собственным нетерминалом:

```
ifunit : if elseif else END
;

if: IF data THEN unit
| IF data THEN
;

elseif: ELSEIF data THEN unit elseif
| ELSEIF data THEN elseif
```

```
| /* empty */
;

else: ELSE unit
| ELSE
| /* empty */
;
```

Листинг 16, однозначная и окончательная грамматика ветвления

Таким образом, мы исправили одну ошибку shift/reduce. Однако есть ещё много неоднозначностей:

```
State 10 conflicts: 3 shift/reduce
State 28 conflicts: 1 reduce/reduce
State 45 conflicts: 4 shift/reduce
State 87 conflicts: 2 shift/reduce
State 188 conflicts: 2 shift/reduce
State 206 conflicts: 2 shift/reduce
```

Листинг 17, оставшиеся конфликты

Проверим в y.output состояние 87:

```
State 87


  64 data: data . OR data
  65     | data . AND data
  66     | data . LESS_THAN data
  67     | data . LESS_EQUAL_THAN data
  68     | data . MORE_THAN data
```

```
 69     | data . MORE_EQUAL_THAN data
 70     | data . TILDE_EQUAL data
 71     | data . EQUALS data
 72     | data . APPEND data
 73     | data . PLUS data
 74     | data . MINUS data
 75     | data . TIMES data
 76     | data . DIVIDE data
 77     | data . MODULO data
 81     | data . POWER data
 82     | data . LEFT_SHIFT data
 83     | data . RIGHT_SHIFT data
 84     | data . PIPE data
 85     | data . AMPERSAND data
 86     | data . TILDE data
111 spot: data .


  PLUS            shift, and go to state 93
  MINUS           shift, and go to state 94
  TIMES           shift, and go to state 95
  DIVIDE          shift, and go to state 96
  POWER           shift, and go to state 97
  MODULO          shift, and go to state 98
  EQUALS          shift, and go to state 99
  LESS_THAN       shift, and go to state 100
  MORE_THAN       shift, and go to state 101
  LESS_EQUAL_THAN  shift, and go to state 102
  MORE_EQUAL_THAN  shift, and go to state 103
  TILDE_EQUAL      shift, and go to state 104
```

```
AND             shift, and go to state 105
OR              shift, and go to state 106
APPEND          shift, and go to state 107
TILDE           shift, and go to state 108
PIPE            shift, and go to state 109
AMPERSAND       shift, and go to state 110
LEFT_SHIFT      shift, and go to state 111
RIGHT_SHIFT     shift, and go to state 112

MINUS     [reduce using rule 111 (spot)]
TILDE     [reduce using rule 111 (spot)]
$default  reduce using rule 111 (spot)
```

Листинг 18, подробности конфликта shift/reduce в состоянии 87.

Как было описано ранее, мы решили вопрос об ассоциативности в пользу левой ассоциации. В этой ситуации подсказка в решении проблемы кроется в “spot: data”. Этот конфликт затрагивает конструктор таблиц, а именно из-за особенностей языка Lua конструктор в таблице фильтрует последнюю незначащую запятую. Например, polyline= {color=4 , tall=7 , }. [2] Дадим описание грамматики обработки параметров, записанных через запятую:

```
hash_table: BRACES_L spotgroup BRACES_R
| BRACES_L BRACES_R
;
spot    : BRACKET_L data BRACKET_R ASSIGN data
| NAME ASSIGN data
| data
;
```

```
spotgroup: next_spotgroup spotdelimiter
;
next_spotgroup: spot
| next_spotgroup spotdelimiter spot
;
spotdelimiter: COMMA
| SEMICOLON
| /*empty*/
;
```

Листинг 19, неудачная грамматика конструктора таблицы стала причиной неоднозначности.

В положении грамматики spotgroup находится нетерминал spotdelimiter, описывающий знаки препинания запятая (COMMA) или точка с запятой (SEMICOLON) или его отсутствие. Воспользуемся приёмом добавления нетерминала, что избавиться от неоднозначности. Добавим нетерминал nextspot. Тогда грамматика принимает вид:

```
hash_table: BRACES_L spotgroup BRACES_R
| BRACES_L BRACES_R
;
spot   : BRACKET_L data BRACKET_R ASSIGN data
| NAME ASSIGN data
| data
;
spotgroup: next_spotgroup nextspot
;
next_spotgroup: spot
```

```
| next_spotgroup spotdelimiter spot
;
nextspot: spotdelimiter
| /* empty */
;

spotdelimiter: COMMA
| SEMICOLON
;
```

Листинг 20, верная грамматика конструктора таблицы.

```
lua.y: warning: 4 shift/reduce conflicts [-Wconflicts-sr]
lua.y: warning: 1 reduce/reduce conflict [-Wconflicts-rr]
```

Рисунок 3, уменьшение конфликтов на данном этапе.

Продолжаем разбор конфликтов:

```
State 10 conflicts: 3 shift/reduce
State 28 conflicts: 1 reduce/reduce
State 45 conflicts: 1 shift/reduce
```

Листинг 21, оставшиеся конфликты, отображенные в y.output.

Второй тип конфликта, который возник при создании грамматики – это reduce/reduce. Этот конфликт возникает, когда на вершине parser стека находится токен, к которому может быть применена свёртка по двум различным правилам в одно и тоже время. Когда это случается, грамматика становится неоднозначной.

Единственный конфликт типа reduce/reduce возник в правилах:

```
instruction   : seqgroup ASSIGN datagroup
| LOCAL namegroup ASSIGN datagroup
| LOCAL namegroup
| FUNCTION name_routine structure_routine
 | LOCAL FUNCTION NAME structure_routine
| jmp_routine
| DO unit END
| DO END
| whileunit
| REPEAT unit UNTIL data
| REPEAT UNTIL data
| ifunit
| forunit
| GOTO NAME
| LABEL
| SEMICOLON
| MYEOF {return 0;}
;
frontdata: seq
| jmp_routine
| PARANTHESES_L data PARANTHESES_R
;
```

Листинг 22, правило, из-за которого возникает reduce/reduce выделено жирным шрифтом и подчёркнуто.

Вывод файла y.output возникшего конфликта:

```
State 28

    instruction: jmp_routine . (rule 15)
    frontdata: jmp_routine . (rule 91)

    STRING          reduce using rule 91 (frontdata)
    DOT             reduce using rule 91 (frontdata)
    COLON           reduce using rule 91 (frontdata)
    BRACES_L        reduce using rule 91 (frontdata)
    BRACKET_L       reduce using rule 91 (frontdata)
    PARANTHESES_L   reduce using rule 15 (instruction)
    PARANTHESES_L   [reduce using rule 91 (frontdata)]
    $default        reduce using rule 15 (instruction)
```

Листинг 23, y.output с состоянием конфликта reduce/reduce.

Здесь правило jmp_routine в некоторых ситуациях может свернуться по правилам instruction и frontdata, что приводит к конфликту. Замена правила jmp_routine на frontdata в правиле instruction позволило исправить конфликт, потому теперь правило достигается только одним путём.

Тогда исправленная грамматика принимает вид:

```
instruction    : seqgroup ASSIGN datagroup
| LOCAL namegroup ASSIGN datagroup
| LOCAL namegroup
| FUNCTION name_routine structure_routine
| LOCAL FUNCTION NAME structure_routine
```

```
| frontdata
| DO unit END
| DO END
| whileunit
| REPEAT unit UNTIL data
| REPEAT UNTIL data
| ifunit
| forunit
| GOTO NAME
| LABEL
| SEMICOLON
| MYEOF {return 0;}
;
```

Листинг 24, исправление конфликта reduce/reduce.

Таким образом, мы добились максимально возможного устранения конфликтов:

```
lua.y: warning: 5 shift/reduce conflicts [-Wconflicts-sr]
```

Рисунок 4, избавление от серьёзного конфликта reduce/reduce.

2.6. Конфликты, которые не удалось разрешить во время создания грамматики

Снова обратимся к y.output файлу для того, чтобы описать оставшиеся неразрешенные конфликты:

```
State 10 conflicts: 3 shift/reduce
State 27 conflicts: 1 shift/reduce
State 45 conflicts: 1 shift/reduce
```

Листинг 25, оставшиеся неразрешенные конфликты.

Всего в работе над парсером осталось 5 конфликтов типа shift/reduce. Три из них связаны с этим правилом:

```
last_piece_code: RETURN datagroup
| RETURN
| BREAK
;
```

Листинг 26, конфликтное правило грамматики.

Проанализировав файл y.output, был сделан вывод, что данный конфликт должен повлиять на логику работы парсера, но так как мы создаём не идеальный интерпретатор, мы можем позволить себе оставить этот конфликт. Вывод y.output:

```
State 10
    last_piece_code: RETURN . datagroup (rule 7)
```

```
                | RETURN . (rule 8)
NIL             shift, and go to state 31
FALSE           shift, and go to state 32
TRUE            shift, and go to state 33
NUMBER          shift, and go to state 34
TDOT            shift, and go to state 35
FUNCTION        shift, and go to state 36
NAME            shift, and go to state 11
MINUS           shift, and go to state 37
STRING          shift, and go to state 38
SQUARE          shift, and go to state 39
NOT             shift, and go to state 40
BRACES_L        shift, and go to state 41
PARANTHESES_L   shift, and go to state 14
TILDE           shift, and go to state 42

FUNCTION        [reduce using rule 8 (last_piece_code)]
NAME            [reduce using rule 8 (last_piece_code)]
PARANTHESES_L   [reduce using rule 8 (last_piece_code)]
$default        reduce using rule 8 (last_piece_code)

seq             go to state 43
data            go to state 60
datagroup       go to state 61
frontdata       go to state 45
routine         go to state 46
jmp_routine     go to state 28
hash_table      go to state 47
```

Листинг 27 , неразрешенный конфликт грамматики в подробностях.

Один конфликт shift/reduce появился после избавления от reduce/reduce, поэтому рассмотрим грамматику instruction с frontdata:

```
instruction   : seqgroup ASSIGN datagroup
| LOCAL namegroup ASSIGN datagroup
| LOCAL namegroup
| FUNCTION name_routine structure_routine
| LOCAL FUNCTION NAME structure_routine
| frontdata
| DO unit END
| DO END
| whileunit
| REPEAT unit UNTIL data
| REPEAT UNTIL data
| ifunit
| forunit
| GOTO NAME
| LABEL
| SEMICOLON
| MYEOF {return 0;}
;
seq: NAME
| frontdata BRACKET_L data BRACKET_R
| frontdata DOT NAME
;
jmp_routine: frontdata args
| frontdata COLON NAME args
;
```

Листинг 28, правило, из-за которого возникает shift/reduce выделено жирным шрифтом и подчёркнуто.

```
State 27

   instruction: frontdata . (rule 15)
   seq: frontdata . BRACKET_L data BRACKET_R  (rule 45)
     | frontdata . DOT NAME  (rule 46)
   jmp_routine: frontdata . args (rule 94)
            | frontdata . COLON NAME args (rule 95)

   STRING         shift, and go to state 70
   DOT            shift, and go to state 71
   COLON          shift, and go to state 72
   BRACES_L       shift, and go to state 41
   BRACKET_L      shift, and go to state 73
   PARANTHESES_L  shift, and go to state 74

   PARANTHESES_L  [reduce using rule 15 (instruction)]
   $default       reduce using rule 15 (instruction)

   args        go to state 75
   hash_table  go to state 76
```

Листинг 29, неразрешенный конфликт положения грамматики instruction в подробностях.

И ещё один конфликт shift/reduce связан со следующим правилами:

seq: NAME
| **frontdata** BRACKET_L data BRACKET_R
| **frontdata** DOT NAME
;
data: NIL
| FALSE
| TRUE
| NUMBER
| STRING
| TDOT
| routine
| **frontdata**
| hash_table
| data OR data
| data AND data
| data LESS_THAN data
| data LESS_EQUAL_THAN data
| data MORE_THAN data
| data MORE_EQUAL_THAN data
| data TILDE_EQUAL data
| data EQUALS data
| data APPEND data
| data PLUS data
| data MINUS data
| data TIMES data
| data DIVIDE data
| data MODULO data
| NOT data

```
| SQUARE data
| MINUS data
| data POWER data
| data LEFT_SHIFT data
| data RIGHT_SHIFT data
| data PIPE data
| data AMPERSAND data
| data TILDE data
| TILDE data
;

jmp_routine: frontdata args
| frontdata COLON NAME args
;
```

Листинг 30, правило, из-за которого возникает shift/reduce выделено жирным шрифтом и подчёркнуто.

```
State 45

   seq: frontdata . BRACKET_L data BRACKET_R (rule 45)
      | frontdata . DOT NAME (rule 46)
   data: frontdata . (rule 62)
   jmp_routine: frontdata . args (rule 94)
            | frontdata . COLON NAME args (rule 95)

   STRING        shift, and go to state 70
   DOT           shift, and go to state 71
   COLON         shift, and go to state 72
```

```
BRACES_L       shift, and go to state 41
BRACKET_L      shift, and go to state 73
PARANTHESES_L  shift, and go to state 74


PARANTHESES_L  [reduce using rule 62 (data)]
$default       reduce using rule 62 (data)


args        go to state 75
hash_table  go to state 76
```

Листинг 31, неразрешенный конфликт грамматик, где задействована frontadata.

3. Интересные регулярные выражения в Lex

3.1. Регулярные выражения вещественных чисел

Как известно в Lua все числа вещественные. Тогда и регулярное выражения будет иметь вид:

```
[0-9]+|[0-9]+[Ee\.][0-9]*|[0-9]*[Ee\.][0-9]+|[0-9]*[\.][0-9]+[Ee][+-]?[0-9]*|
[0-9]+[\.][Ee][+-]?[0-9]*|0x[A-Fa-f0-9]+|0x[A-Fa-f0-9]+"."[A-Fa-f0-9]*
```

Листинг 32, регулярные выражения вещественных чисел.

Разберем по порядку регулярные выражения представления вещественных чисел:

- [0-9]+ Совпадение хотя бы одного любой цифры
- [0-9]+[Ee\.][0-9]* Совпадение вещественных чисел с прописной или заглавной экспонентой или с наличием точки, а после экспоненты или точки могут следовать или не следовать цифры. Например, 1235E; 1234e; 1235. ; 1235E7 ; 1235e8 ; 1235.9

- [0-9]*[Ee\.][0-9]+ Число может начинаться с цифр или нет. Но затем обязательно должна следовать экспонента (прописная или заглавная) или точка, а после экспоненты или точки должна следовать как минимум одна цифра. Например, e748; E748; .748 ; 1235E7 ; 1235e748 ; 12345.1

- [0-9]*[\.][0-9]+[Ee][+-]?[0-9]* Число может начинаться с цифр или нет. Но затем обязательно должна следовать точка. Затем обязательно должна следовать как минимум одна цифра. После должна следовать прописная или заглавная экспонента, затем необязательный плюс или минус, и необязательная цифра. Например, 123.2E ; 123.2e ; .23e ; .24E ; 123.2E+5 ; 123.2e-5

- [0-9]+[\.][Ee][+-]?[0-9]* Число должно иметь в начале как минимум одну цифру , затем обязательная точка , затем обязательная прописаная или заглавная экспонента, затем необязательный плюс или минус и необязательная цифра. Например, 123.e99 ; 123.E-99

- 0x[A-Fa-f0-9]+ Шестнадцатеричные числа обязательно начинаются на 0x , а затем могут идти заглавные или прописные латинские символы или числа один или более раз

- 0x[A-Fa-f0-9]+"."[A-Fa-f0-9]* Шестнадцатеричные числа обязательно начинаются на 0x , а затем могут идти заглавные или прописные латинские символы один или более раз. Затем идёт обязательная точка и не обязательные заглавные или прописные латинские символы или числа.

3.2. Регулярные выражения поиска строк

Для поиска строк использовались следующие регулярные выражения:

\"([^\\\"] | \\\n | \\.)*\"

\'([^\\\'] | \\\n | \\.)*\'

Они отличаются только кавычками, так как Lua позволяет записывать строки двумя видами кавычек.

Мы экранируем кавычки по бокам, чтобы происходило совпадение со строками в соответствии с правилами Lua. В первом случае принимаются любые символы, кроме слеша и кавычки. Для нас опасен вариант, если придут две слеша (\\), поэтому мы упрощаем регулярное выражение тем, что после слеш (третье правило) следует любой символ. Регулярное выражение съедает хотя бы один символ слэш за тем, чтобы, откатываясь назад, алгоритм поиска не смог разбить \\ на два вхождения \. Так же вынесен перенос строки в отдельное правило, так как на точку \n нет совпадения.

3.3. Парсинг комментариев Lua

Для совпадения однострочных комментариев используется следующее регулярное выражение:

- "--".* Ищется совпадение двух тире и затем любое количество вхождений любых символов, кроме \n

Для совпадения много строчных комментариев (с равенством и без):

```
%x LUA_COMMENT
"--\[\[" { BEGIN(LUA_COMMENT); }
"--\[=".* {BEGIN(LUA_COMMENT); }
<LUA_COMMENT>"\]\]" {BEGIN(INITIAL);}
<LUA_COMMENT>"=\]" {BEGIN(INITIAL);}
<LUA_COMMENT>. {}
<LUA_COMMENT>\n {lines++;}
```

Листинг 33, парсинг комментариев.

В этом подходе использовано start condition. Определение %x с пометкой LUA_COMMENT создаёт эксклюзивное состояние, которое означает, что лексер будет искать совпадения в комментарии только с лексемами, помеченными LUA_COMMENT.

%x LUA_COMMENT определяет состояние под названием LUA_COMMENT. И правила "--\[\[" и "--\[=".* определяют начало многострочного комментария. Как только найдётся совпадение с началом комментария, лексемы конца комментария ("\]\]" и "=\]") перейдут в INITIAL состояние. И теперь когда оба правила (начала и конца) совпали, внутренности комментария совпадают только с лексемами,помеченными метками LUA_COMMENT. Помеченная точка съедает все символы кроме переноса строки, а перенос строки инкрементирует глобальный счётчик количества строк. [3]

Заключение

В результате выполнения поставленных задач, мы создали хоть и не идеальный, но вполне работоспособный парсер. Далее мы сделаем вывод и ответим на главный вопрос этой работы: “Насколько YACC соответствует современному средству создания компиляторов?”.

YACC соответствует современному средству создания компиляторов лишь частично. Такие языки как Go и Ruby содержат грамматику YACC, что означает, что они могут быть скомпилированы, используя Lex и YACC (однако длина листинга достигает 11 000 строк), а также и PHP, и Perl5. Другой факт, что GCC до версии 3.4 использовал в своём составе парсер YACC.

На своём примере мы показали, что создать парсер с восходящим алгоритмом синтаксического разбора (LALR(1)) можно, но это будет очень утомительным занятием. Единственным сильным аргументом в пользу генератора парсеров является возможность самодокументирования. Самодокументирование имеется в виду в том смысле, что грамматика, использованная для генерации синтаксического анализатора, является именно той грамматикой, которую можно поместить в документацию к языку.

Далее будут приведены аргументы и примеры, в пользу грамматик рекурсивного спуска.

Как упоминалось выше написание парсера вручную – это сложная работа, но гораздо более серьёзная проблема – возможность компилятором проводить хорошую оптимизацию. А для это не существует простых инструментов у генератора парсеров.

GCC с версии 3.5 заменил YACC на парсер рекурсивного спуска, написанный вручную, потому что его было трудно поддерживать и расширять. Это дало ускорение в 1.5% при компиляции языка C. [4]

Более того грамматики рекурсивного спуска позволяют создавать очень хорошие сообщения об ошибках, тогда как в LALR для этого нужно постараться. А хорошие сообщения об ошибках очень важны для восприятия языка.

Зададимся вопросом: можно ли использовать YACC и Lex, чтобы написать компилятор C++?

Первая же проблема, которая при этом возникает, это зависимость от контекста. Скорее всего придётся создать специальную таблицу, к которой будет обращаться парсер, и которая будет отвечать ему на вопрос “А что означает этот идентификатор в данном контексте?”. И такая таблица будет включать в себя сотни положений стандарта C++. Например, “foo * bar;”
Может быть распознан как умножение, либо как объявление переменной типа указатель. Даже если эти проблемы решаемы, то остальная часть работы компилятора, например, препроцессор, проверка типов данных, генерация кода, оптимизации и распределение регистров и так далее будут очень трудоёмкими. Таким образом, мы приходим к ответу, что YACC – это просто не подходящий инструмент для создания C++ компилятора.

На месте языка C++ может быть любой другой современный и сложный язык. Написать парсер для такого языка хоть и очень сложно и, вероятно, это будет “сизифов труд”, но возможно.

Список использованных источников

1. Иерусалимский Р. Программирование на Lua: учебное пособие / Иерусалимский Р. М: Изд-во ДМК-Пресс, 2010.-15 с.

2. Revi A. Parse Stack, //silcnitc.github.io: debug parsing stack. – 2015. URL: https://silcnitc.github.io/index.html - (дата обращения: 16.05.2020).

3. Левин Д. flex&bison: учебное пособие / Левин Д., Браун Д. – США: Изд-во O'REILY, 2009. – 128 с.

4. Майерс Д. New C Parser, //gcc.gnu.org: New C Parser. – 2008. URL: https://gcc.gnu.org/wiki/New_C_Parser - (дата обращения: 3.06.2020).

Приложение

```
%%

unit : piece_code {printf("unit is here\n");}
;

piece_code : futher_piece_code last_piece_code {printf("piece_code is here\n");}
| futher_piece_code {printf("piece_code is here\n");}
| last_piece_code {printf("piece_code is here\n");}
;

futher_piece_code: instruction {printf("futher_piece_code is here\n");}
| piece_code instruction {printf("futher_piece_code is here\n");}
;

last_piece_code: RETURN datagroup {printf("last_piece_code is here\n");}
| RETURN {printf("last_piece_code is here\n");}
| BREAK {printf("last_piece_code is here\n");}
;


instruction   : seqgroup ASSIGN datagroup {printf("instruction is here\n");}
| LOCAL namegroup ASSIGN datagroup {printf("instruction is here\n");localVars++;}
| LOCAL namegroup {printf("instruction is here\n");localVars++;}
| FUNCTION name_routine structure_routine {printf("instruction is here\n");functionCount++;}
| LOCAL FUNCTION NAME structure_routine {printf("instruction is here\n");functionCount++;localFunctions++;}
| jmp_routine {printf("instruction is here\n");functionCalls++;}
```

```
| DO unit END {printf("instruction is here\n");doUnits++;}
| DO END {printf("instruction is here\n");doUnits++;}
| whileunit {printf("instruction is here\n");whileUnits++;}
| REPEAT unit UNTIL data {printf("instruction is here\n");repeatUnits++;}
| REPEAT UNTIL data {printf("instruction is here\n");repeatUnits++;}
| ifunit {printf("instruction is here\n");ifUnits++;}
| forunit {printf("instruction is here\n");forUnits++;}
| GOTO NAME {printf("instruction is here\n");}
| LABEL {printf("instruction is here\n");}
| SEMICOLON
| MYEOF {return 0;}
;

forunit: FOR NAME ASSIGN data COMMA data DO unit END {printf("forunit is here\n");}
| FOR NAME ASSIGN data COMMA data COMMA data DO unit END {printf("forunit is here\n");}
| FOR namegroup IN datagroup DO unit END {printf("forunit is here\n");}
| FOR NAME ASSIGN data COMMA data DO END {printf("forunit is here\n");}
| FOR NAME ASSIGN data COMMA data COMMA data DO END {printf("forunit is here\n");}
| FOR namegroup IN datagroup DO END {printf("forunit is here\n");}
;

whileunit: WHILE data DO unit END {printf("whileunit is here\n");}
| WHILE data DO END {printf("whileunit is here\n");}
;


ifunit: IF data THEN unit else {printf("if is here\n");}
```

```
;
else: END
|ELSE unit END
|elseifs
;
elseifs: elseifs ELSEIF data THEN unit else {printf("elseif\n");}
|ELSEIF data THEN unit else{printf("Multiple amount of elseif\n");}
;


seq: NAME {printf("seq is here\n");}
| frontdata BRACKET_L data BRACKET_R {printf("seq is here\n");}
| frontdata DOT NAME {printf("seq is here\n");}
;

seqgroup: seq {printf("seqgroup is here\n");}
| seqgroup COMMA seq {printf("seqgroup is here\n");}
;

name_routine: next_name_routine {printf("name_routine is here\n");}
| next_name_routine COLON NAME {printf("name_routine is here\n");}
;

next_name_routine: NAME {printf("next_name_routine is here\n");}
| next_name_routine DOT NAME {printf("next_name_routine is here\n");}
;

namegroup: NAME {printf("namegroup is here\n");}
| namegroup COMMA NAME {printf("namegroup is here\n");}
                ;
```

```
data: NIL {printf("data is here\n");}
| FALSE {printf("data is here\n");}
| TRUE {printf("data is here\n");}
| NUMBER {printf("data is here\n");}
| STRING {printf("data is here\n");}
| TDOT {printf("data is here\n");}
| routine {printf("data is here\n");}
| frontdata {printf("data is here\n");}
| hash_table {printf("data is here\n");}
| data OR data {printf("data is here\n");}
| data AND data {printf("data is here\n");}
| data LESS_THAN data {printf("data is here\n");}
| data LESS_EQUAL_THAN data {printf("data is here\n");}
| data MORE_THAN data {printf("data is here\n");}
| data MORE_EQUAL_THAN data {printf("data is here\n");}
| data TILDE_EQUAL data {printf("data is here\n");}
| data EQUALS data {printf("data is here\n");}
| data APPEND data {printf("data is here\n");}
| data PLUS data {printf("data is here\n");}
| data MINUS data {printf("data is here\n");}
| data TIMES data {printf("data is here\n");}
| data DIVIDE data {printf("data is here\n");}
| data MODULO data {printf("data is here\n");}
| NOT data {printf("data is here\n");}
| SQUARE data {printf("data is here\n");}
| MINUS data {printf("data is here\n");}
| data POWER data {printf("data is here\n");}
| data LEFT_SHIFT data {printf("data is here\n");}
| data RIGHT_SHIFT data {printf("data is here\n");}
```

```
| data PIPE data {printf("data is here\n");}
| data AMPERSAND data {printf("data is here\n");}
| data TILDE data {printf("data is here\n");}
| TILDE data {printf("data is here\n");}

;

datagroup : data {printf("datagroup is here\n");}
| datagroup COMMA data {printf("datagroup is here\n");}
;

frontdata: seq {printf("frontdata is here\n");}
| jmp_routine {printf("frontdata is here\n");}
| PARANTHESES_L data PARANTHESES_R {printf("frontdata is here\n");}
;


routine: FUNCTION structure_routine {printf("routine is here\n");}
;

jmp_routine: frontdata args {printf("jmp_routine is here\n");}
| frontdata COLON NAME args {printf("jmp_routine is here\n");}
;

structure_routine: PARANTHESES_L parameters PARANTHESES_R unit END {printf("structure_routine is here\n");}
| PARANTHESES_L PARANTHESES_R unit END {printf("structure_routine is here\n");}
| PARANTHESES_L PARANTHESES_R END {printf("structure_routine is here\n");}
```

```
| PARANTHESES_L parameters PARANTHESES_R END
{printf("structure_routine is here\n");}
;

parameters   : namegroup {printf("parameters is here\n");}
| namegroup COMMA TDOT {printf("parameters is here\n");}
| TDOT {printf("parameters is here\n");}
;

args   : PARANTHESES_L PARANTHESES_R {printf("args is here\n");}
| PARANTHESES_L datagroup PARANTHESES_R {printf("args is here\n");}
| hash_table {printf("args is here\n");}
| STRING {printf("args is here\n");}
;




hash_table: BRACES_L spotgroup BRACES_R {printf("hash_table is here\n");}
| BRACES_L BRACES_R {printf("hash_table is here\n");}
;

spot   : BRACKET_L data BRACKET_R ASSIGN data {printf("spot is here\n");}
| NAME ASSIGN data {printf("spot is here\n");}
| data {printf("spot is here\n");}
;

spotgroup: next_spotgroup nextspot {printf("spotgroup is here\n");}
```

```
;

next_spotgroup: spot {printf("next_spotgroup is here\n");}
| next_spotgroup spotdelimiter spot {printf("next_spotgroup is here\n");}

nextspot: spotdelimiter {printf("nextspot is here\n");}
| /* empty */ {printf("nextspot is here\n");}
;

spotdelimiter: COMMA {printf("spotdelimiter is here\n");}
| SEMICOLON {printf("spotdelimiter is here\n");}
;


%%
```