*Федеральное государственное бюджетное образовательное учреждение*

*высшего профессионального образования*

|  |  |
| --- | --- |
|  | ***«Московский государственный технический университет  имени Н.Э. Баумана»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

**РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовой работе на тему:**

**Компилятор языка Lua на языке Python.**

Студент И. В. Спасенов

(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. А. Ступников

(Подпись, дата) (И.О.Фамилия)

Москва, 2021

[**Введение**](#_sb8m9brhlpze) **4**

1. [**Аналитический раздел**](#_cru651yzr2l6) **5**
   1. [Лексический и синтаксический анализ](#_leok054ygbjn) 5
   2. [Методы реализации лексического и синтаксического анализаторов](#_g2m6e8lof97t) 6
      1. [Алгоритмы лексического и синтаксического анализа](#_i3d9rlvz78e8) 6
      2. [Стандартные средства](#_ajbtjwiufuh5) 6
   3. [Генератор ANTLR](#_aupl0qnjpw8l) 7
   4. [Построение входной грамматики в формате ANTLR](#_ljyvtdjq32bs) 7
   5. [Семантический анализ](#_ow98fb5fmh4c) 8
      1. [Обнаружение и обработка семантических ошибок](#_6sg7odcs01qd) 8
   6. [Генерация кода](#_c1s9x1o1qp5y) 9
2. [**Конструкторский раздел**](#_2ka2y1j55xva) **10**
   1. [Структура компилятора](#_9cgqj0qpj2jn) 10
   2. [Генерируемые классы](#_8u7ijrs631e) 10
   3. [Получение дерева разбора](#_2ifm87iijxfz) 11
   4. [Обнаружение и обработка лексических и синтаксических ошибок](#_gcyogky5fq1a) 11
   5. [Обход дерева разбора](#_47lc898ex73g) 11
   6. [Генерация кода](#_i4ozsssduzsn) 12
3. [**Использование компилятора**](#_iaws06f5soiw) **14**
   1. [Пример компиляции программы](#_baokkdoeu8mz) 14

[**Заключение**](#_gq9l1yotxvpm) **17**

[**Список литературы**](#_dcw60m76zjbl) **18**

[**Приложение А**](#_ebzjcs9ptay6) **19**

# Введение

Цель работы — разработать приложение на языке Python, выполняющее лексический и синтаксический анализ кода на языке Lua, построение графа вызовов функций, а также таблиц локальных и глобальных переменных для локальных и глобальных функций.

Для достижения цели работы необходимо решить следующие задачи:

1. Разработать блок лексического и синтаксического анализа с построением дерева разбора для заданного исходного кода языка Lua ;
2. Разработать блок семантического анализа.

# Аналитический раздел

## Лексический и синтаксический анализ

Задачей лексического анализа является аналитический разбора входной последовательности символов составляющих текст компилируемой программы с целью получения на выходе последовательности символов, называемых «токенами», которые характеризуются определенными типом и значением.

Лексический анализатор функционирует в соответствии с некоторыми правилами построения допустимых входных последовательностей. Данные правила могут быть определены, например, в виде детерминированного конечного автомата, регулярного выражения или праволинейной грамматики. С практической точки зрения наиболее удобным способом является формализация работы лексического анализатора с помощью грамматики.

Лексический анализ может быть представлен и как самостоятельная фаза трансляции, и как составная часть фазы синтаксического анализа. В первом случае лексический анализатор реализуется в виде отдельного модуля, который принимает последовательность символов, составляющих текст компилируемой программы, и выдаёт список обнаруженных лексем. Во втором случае лексический анализатор фактически является подпрограммой, вызываемой синтаксическим анализатором для получения очередной лексемы [2].

В процессе лексического анализа обнаруживаются лексические ошибки – простейшие ошибки компиляции, связанные с наличием в тексте программы недопустимых символов, некорректной записью идентификаторов, числовых констант и пр.

Синтаксический анализ, или разбор, как его еще называют, – это процесс сопоставления линейной последовательности токенов исходного языка с его формальной грамматикой. Результатом обычно является дерево разбора (или абстрактное синтаксическое дерево или конкретное синтаксическое дерево).

Синтаксический анализатор фиксирует синтаксические ошибки, т.е. ошибки, связанные с нарушением принятой структуры программы.

## Методы реализации лексического и синтаксического анализаторов

Лексический и синтаксический анализаторы могут быть разработаны «с нуля» на основе существующих алгоритмов анализа, а могут быть созданы с помощью стандартных средств генерации анализаторов.

### Алгоритмы лексического и синтаксического анализа

Существуют две основные стратегии синтаксического анализа: нисходящий анализ и восходящий анализ.

В нисходящем анализе дерево вывода цепочки строится от корня к листьям, т.е. дерево вывода «реконструируется» в прямом порядке, и аксиома грамматики «развертывается» в цепочку. В общем виде нисходящий анализ представлен в анализе методом рекурсивного спуска, который может использовать откаты, т.е. производить повторный просмотр считанных символов [1].

В восходящем анализе дерево вывода строится от листьев к корню и анализируемая цепочка «сворачивается» в аксиому. На каждом шаге свертки некоторая подстрока, соответствующая правой части продукции, замещается левым символом данной продукции. Примерами восходящих синтаксических анализаторов являются синтаксические анализаторы приоритета операторов, LR-анализаторы (SLR, LALR) [1].

### Стандартные средства

Имеется множество различных стандартных средств для построения синтаксических анализаторов: Lex и Yacc, Coco/R, ANTLR, JavaCC и др.

Генератор Yacc предназначен для построения синтаксического анализатора контекстно-свободного языка. Анализируемый язык описывается с помощью грамматики в виде, близком форме Бэкуса-Наура. Результатом работы Yacc'a является программа на Си, реализующая восходящий LALR(1) распознаватель. Как правило, Yacc используется в связке с Lex – стандартным генератором лексических анализаторов. Для обоих этих инструментов существуют свободные реализации – Bison и Flex.

Сосо/R читает файл с атрибутивной грамматикой исходного языка в расширенной форме Бэкуса – Наура и создает файлы лексического и синтаксического анализаторов. Лексический анализатор работает как конечный автомат. Синтаксический анализатор использует методику нисходящего рекурсивного спуска.

ANTLR (ANother Tool for Language Recognition) – это генератор синтаксических анализаторов для чтения, обработки или трансляции как структурированных текстовых, так и бинарных файлов. ANTLR широко используется для разработки компиляторов, прикладных программных инструментов и утилит. На основе заданной грамматики языка ANTLR генерирует код синтаксического анализатора, который может строить абстрактное синтаксического дерево и производить его обход.

Принимая во внимание эффективность и простоту использования ANTLR, для построения кода синтаксического анализатора было решено применить данное средство.

## Генератор ANTLR

В качестве входных данных для ANTLR выступает файл с описанием грамматики исходного языка. Данный файл содержит только правила грамматики без добавления кода, исполнение которого соответствует применению определённых правил. Подобное разделение позволяет использовать один и тот же файл грамматики для построения различных приложений (например, компиляторов, генерирующих код для различных сред исполнения).

На основе правил заданной грамматики языка ANTLR генерирует класс нисходящего рекурсивного синтаксического анализатора. Для каждого правила грамматики в полученном классе имеется свой рекурсивный метод. Разбор входной последовательности начинается с корня синтаксического дерева и заканчивается в листьях.

## Построение входной грамматики в формате ANTLR

При использовании ANTLR заданную грамматику языка необходимо записать в принятом входном формате. Построение правил грамматик ANTLR происходит на основе следующих шаблонов [3]:

1. **Последовательность**: несколько подряд идущих элементов (например, значения, указываемые при инициализации массива);
2. **Выбор**: несколько возможных альтернатив (выражение блока может быть вызовом функции write или writeln, или представлять собой операцию присваивания, или так же являться блоком и т.п.);
3. **Контекст** (зависимость лексем): присутствие в тексте программы одной лексемы требует наличия другой (например, для каждой левой открывающей круглой скобки должна быть соответствующая закрывающая правая);
4. **Вложенные конструкции**: однотипные рекурсивно определяемые выражения (арифметические выражения, вложенные блоки, циклы в цикле и т.д.).

## Семантический анализ

В процессе семантического анализа происходит накопление данных о переменных, функциях и используемых типах для генерации кода. Информация об используемых объектах сохраняется в иерархические структуры данных т.н. таблицы символов.

### Обнаружение и обработка семантических ошибок

Задача компилятора – проверить, что исходная программа соответствует принятым синтаксическим и семантическим соглашениям заданного языка. Такая проверка, именуемая статической, – в отличие от динамической, выполняемой в процессе работы целевой программы, – обеспечивает, что будут обнаружены определённые типы программных ошибок. Принято разделять следующие виды статических проверок [1]:

1. Проверки типов: компилятор должен сообщать об ошибке, если оператор применяется к несовместимому с ним операнду;
2. Проверки управления: передача управления за пределы языковых конструкций должна осуществляться в определённое место;
3. Проверки единственности и существования: могут быть ситуации, когда в некотором контексте, определенный идентификатор может или должен встречаться не более одного раза.

Неуспешное завершение каждой из проверок означает семантическую ошибку.

# Конструкторский раздел

## В рамках работы было создано консольное приложение, написанное на языке Python, которое в качестве параметра принимает путь к файлу компилируемой программы и генерирует граф вызовов функции, таблицы локальных, глобальных переменных для локальных и глобальных функций.

## **Генерируемые классы**

В данной работе использовался ANTLR версии 4, так как она является последней на сегодняшний день.

Исходная грамматика, записанная в формате ANTLR (Приложение А), указывается в текстовом файле, обрабатывая который, ANTLR генерирует ряд классов анализаторов.

Для файла Lua.g4, содержащего описание грамматики языка Lua генерируются следующие файлы [3]:

* LuaParser.py;
* LuaLexer.py;
* LuaListener.py;
* LuaLexer.tokens.

Файл LuaParser.py содержит определение синтаксического анализатора для проверки структуры программы и построения дерева разбора. Для каждой альтернативы каждого правила грамматики в данном классе определен свой метод.

В LuaLexer.py находится сгенерированный класс лексического анализатора, который разбивает входной поток символов на отдельные лексемы, исходя из определений терминалов в исходной грамматике.

LuaListener.py содержит виртуальные методы, которые вызываются при обходе дерева разбора, полученного от синтаксического анализатора.

## Получение дерева разбора

Сгенерированный ANTLR синтаксический анализатор выдаёт конкретное синтаксическое дерево, и реализует методы для его построения и последующего обхода. Дерево разбора для заданной входной последовательности символов можно получить, вызвав метод, соответствующий аксиоме в исходной грамматике языка. В грамматике языка Lua аксиомой является нетерминал block, поэтому построение дерева следует начинать с вызова метода block() объекта класса синтаксического анализатора, являющейся корнем дерева.

## Обход дерева разбора

Для обхода синтаксического дерева используется реализация класса LuaListener. Базовый класс содержит методы входа и выхода в узлы различного типа, полученные из грамматики Lua.g4. При этом в методы передается текущий рассматриваемый узел с информацией о родителе этого узла, а также о его потомках. Были реализованы методы класса LuaListener для сохраненения информации о глобальных и локальных переменных, глобальных и локальных фукциях и их вызовах.

## Пример работы программы

Рассмотрим пример компиляции Lua программы. На рисунке 1 приведен пример программы на языке Lua, на ней будет видна работа системы с локальными и глобальными функциями, вложенными функциями, локальными и глобальными переменными.

|  |
| --- |
| function fact (n)  if n == 0 then  return 1  else  return n \* fact(n-1)  end  end  function func4 (n)  for i = 1, n do  print("func4")  end  end  function func3 (n)  local function func3\_inner\_function(n)  return n + 1  end  print("func3")  b, c = 5, 8  b = func3\_inner\_function(b)  func4(b + c + n)  end  function func2 (n)  for i = 1, n % 3 do  print("func2")  func3(i\*n/3)  end  end  function func1 (n)  print("func1")  n = n + 2  func2(n)  end  function square(num)  res = num \* num  return res  end  local function sub(num1, num2)  local res = num1 - num2  return res  end  function add(num1, num2)  b5 = square(num1)  b6 = fact(num2)  res = b5 + b6  return res  end  a, b, c = 9, 8  a = 5  res\_func = func1(a)  res\_add = add(b, c)  local a1, a2, a3 = 5, 2, 1  b = 'a("bbbbb")'  o = {}  o.n1, o.n2 = 1, 2  o.str1, o['str2'] = {1,2,3}, {4,5,6}  o['str1'][3] = 100  o['str1'][30] = 200  o['str2'][3] = {'nn', 10, 20, {x = 5, y = 7}} |
| Рис 1. Пример программы на языке Lua. |

На рисунке 2 показан граф вызовов функций написанной программы, в узлах грфа находятся имена функций, в следующем формате: «(имя функции, имя функции в области видимости которой находится функция) [аргументы функции]», стрелками показаны вызовы функций. Символом «\_G» обозначена базовая функция из которой вызываются остальные, считается, что любая функция вызываемая вне остальных функций выхывается «\_G»

|  |
| --- |
|  |
| Рис 2. Граф вызовов |

На рисунках 3, 4 показаны таблицы глобальных и локальны переменных для глобальных и локальных функций соответственно. Для каждой переменной указывается ее тип: number – число, string – строка, table – таблица, func – переменная является значением функции, ее тип не определен, expression – переменная является значением выражения, ее тип не определен, nill – переменная не определена. Также указывается ее последнее значение.

|  |
| --- |
|  |
| Рис 3. Таблица переменных для глобальных функций |

|  |
| --- |
|  |
| Рис 4. Таблица переменных для локальных функций |

На рисунке 5 приведен пример реализации алгоритма быстрой сортировки на языке Lua.

|  |
| --- |
| function partition(array, left, right, pivotIndex)  local pivotValue = array[pivotIndex]  array[pivotIndex], array[right] = array[right], array[pivotIndex]    local storeIndex = left    for i = left, right-1 do  if array[i] <= pivotValue then  array[i], array[storeIndex] = array[storeIndex], array[i]  storeIndex = storeIndex + 1  end  array[storeIndex], array[right] = array[right], array[storeIndex]  end    return storeIndex  end  function quicksort(array, left, right)  if right > left then  local pivotNewIndex = partition(array, left, right, left)  quicksort(array, left, pivotNewIndex - 1)  quicksort(array, pivotNewIndex + 1, right)  end  end  array = { 1, 5, 2, 17, 11, 3, 1, 22, 2, 37 }  quicksort(array, 1, 10)  print(array) |
| Рис 5. Пример программы на языке Lua. |

На рисунке 6 показан граф вызовов функций написанной программы

|  |
| --- |
|  |
| Рис 6. Граф вызовов |

На рисунке 7 показана таблицы глобальных и локальны переменных для глобальных функций, локальных функций в программе нет.

|  |
| --- |
|  |
| Рис 3. Таблица переменных для глобальных функций |

# Заключение

В рамках курсовой работы был приведен обзор основных алгоритмов лексического и синтаксического анализа. Были рассмотрены стандартные средства построения синтаксических анализаторов. Описано использование пакета прикладных программ ANTLR4 для генерации исходного кода классов анализаторов по заданной грамматике языка Lua. Была разработана программа на языке Python, выполняюшая, построение графа вызовов функций, а также таблиц локальных и глобальных переменных для локальных и глобальных функций.

# Список литературы

1. Ахо А., Сети Р., Ульман Д. «Компиляторы. Принципы, технологии, инструменты». — М.: Вильямс, 2001.
2. Серебряков В. А., Галочкин М. П. «Основы конструирования компиляторов».
3. Terence Parr «Definitive ANTLR4 reference». — М.: Pragmatic Bookshelf, 2013.

# Приложение А

Грамматика языка Lua в формате ANTLR.

|  |
| --- |
| grammar Lua;  chunk  : block EOF  ;  block  : stat\* retstat?  ;  stat  : ';'  | varlist '=' explist  | functioncall  | label  | 'break'  | 'goto' NAME  | 'do' block 'end'  | 'while' exp 'do' block 'end'  | 'repeat' block 'until' exp  | 'if' exp 'then' block ('elseif' exp 'then' block)\* ('else' block)? 'end'  | 'for' NAME '=' exp ',' exp (',' exp)? 'do' block 'end'  | 'for' namelist 'in' explist 'do' block 'end'  | 'function' funcname funcbody  | 'local' 'function' NAME funcbody  | 'local' attnamelist ('=' explist)?  ;  attnamelist  : NAME attrib (',' NAME attrib)\*  ;  attrib  : ('<' NAME '>')?  ;  retstat  : 'return' explist? ';'?  ;  label  : '::' NAME '::'  ;  funcname  : NAME ('.' NAME)\* (':' NAME)?  ;  varlist  : var\_ (',' var\_)\*  ;  namelist  : NAME (',' NAME)\*  ;  explist  : exp (',' exp)\*  ;  exp  : 'nil' | 'false' | 'true'  | number  | string  | '...'  | functiondef  | prefixexp  | tableconstructor  | <assoc=right> exp operatorPower exp  | operatorUnary exp  | exp operatorMulDivMod exp  | exp operatorAddSub exp  | <assoc=right> exp operatorStrcat exp  | exp operatorComparison exp  | exp operatorAnd exp  | exp operatorOr exp  | exp operatorBitwise exp  ;  prefixexp  : varOrExp nameAndArgs\*  ;  functioncall  : varOrExp nameAndArgs+  ;  varOrExp  : var\_ | '(' exp ')'  ;  var\_  : (NAME | '(' exp ')' varSuffix) varSuffix\*  ;  varSuffix  : nameAndArgs\* ('[' exp ']' | '.' NAME)  ;  nameAndArgs  : (':' NAME)? args  ;  args  : '(' explist? ')' | tableconstructor | string  ;  functiondef  : 'function' funcbody  ;  funcbody  : '(' parlist? ')' block 'end'  ;  parlist  : namelist (',' '...')? | '...'  ;  tableconstructor  : '{' fieldlist? '}'  ;  fieldlist  : field (fieldsep field)\* fieldsep?  ;  field  : '[' exp ']' '=' exp | NAME '=' exp | exp  ;  fieldsep  : ',' | ';'  ;  operatorOr  : 'or';  operatorAnd  : 'and';  operatorComparison  : '<' | '>' | '<=' | '>=' | '~=' | '==';  operatorStrcat  : '..';  operatorAddSub  : '+' | '-';  operatorMulDivMod  : '\*' | '/' | '%' | '//';  operatorBitwise  : '&' | '|' | '~' | '<<' | '>>';  operatorUnary  : 'not' | '#' | '-' | '~';  operatorPower  : '^';  number  : INT | HEX | FLOAT | HEX\_FLOAT  ;  string  : NORMALSTRING | CHARSTRING | LONGSTRING  ;  // LEXER  NAME  : [a-zA-Z\_][a-zA-Z\_0-9]\*  ;  NORMALSTRING  : '"' ( EscapeSequence | ~('\\'|'"') )\* '"'  ;  CHARSTRING  : '\'' ( EscapeSequence | ~('\''|'\\') )\* '\''  ;  LONGSTRING  : '[' NESTED\_STR ']'  ;  fragment  NESTED\_STR  : '=' NESTED\_STR '='  | '[' .\*? ']'  ;  INT  : Digit+  ;  HEX  : '0' [xX] HexDigit+  ;  FLOAT  : Digit+ '.' Digit\* ExponentPart?  | '.' Digit+ ExponentPart?  | Digit+ ExponentPart  ;  HEX\_FLOAT  : '0' [xX] HexDigit+ '.' HexDigit\* HexExponentPart?  | '0' [xX] '.' HexDigit+ HexExponentPart?  | '0' [xX] HexDigit+ HexExponentPart  ;  fragment  ExponentPart  : [eE] [+-]? Digit+  ;  fragment  HexExponentPart  : [pP] [+-]? Digit+  ;  fragment  EscapeSequence  : '\\' [abfnrtvz"'\\]  | '\\' '\r'? '\n'  | DecimalEscape  | HexEscape  | UtfEscape  ;  fragment  DecimalEscape  : '\\' Digit  | '\\' Digit Digit  | '\\' [0-2] Digit Digit  ;  fragment  HexEscape  : '\\' 'x' HexDigit HexDigit  ;  fragment  UtfEscape  : '\\' 'u{' HexDigit+ '}'  ;  fragment  Digit  : [0-9]  ;  fragment  HexDigit  : [0-9a-fA-F]  ;  COMMENT  : '--[' NESTED\_STR ']' -> channel(HIDDEN)  ;  LINE\_COMMENT  : '--'  ( // --  | '[' '='\* // --[==  | '[' '='\* ~('='|'['|'\r'|'\n') ~('\r'|'\n')\* // --[==AA  | ~('['|'\r'|'\n') ~('\r'|'\n')\* // --AAA  ) ('\r\n'|'\r'|'\n'|EOF)  -> channel(HIDDEN)  ;  WS  : [ \t\u000C\r\n]+ -> skip  ;  SHEBANG  : '#' '!' ~('\n'|'\r')\* -> channel(HIDDEN)  ; |