

Федеральное агентство по образованию
Государственное 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»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе по курсу
Конструирование компиляторов
Тема: «Компилятор языка Lua»

Студент: Донцов В. В.

Группа: ИУ7-27

Научный руководитель: Просуков Е. А.

Оглавление

Введение	4
1. Лексический и синтаксический анализ	6
1.1. Реализация лексического и синтаксического анализаторов	7
1.1.1. Лексические соглашения	7
1.1.2. Блоки	10
1.1.3. Присваивания	10
1.1.4. Циклы for, while, repeat	12
1.1.5. Локальные объявления	13
1.1.6. Конструктор таблиц	13
1.1.7. Вызовы функций	14
1.2. Получение дерева разбора	16
2. Семантический анализ	18
3. Тестирование	21
4. Приложение	24
Заключение	25
Список используемой литературы	26

Введение

Lua — скриптовый язык программирования, разработанный в подразделении Tecgraf (Computer Graphics Technology Group) Католического университета Рио-де-Жанейро (Бразилия). По возможностям, идеологии и реализации язык ближе всего к JavaScript, однако Lua отличается более мощными и гораздо более гибкими конструкциями.

Язык широко используется для создания тиражируемого программного обеспечения (например, на нём написан графический интерфейс пакета Adobe Lightroom). Также получил известность как язык программирования уровней и расширений во многих играх из-за удобства встраивания, скорости исполнения кода и лёгкости обучения.

Историческими родителями Lua были языки конфигурирования и описания данных SOL (Simple Object Language) и DEL (Data-Entry Language). Они были независимо разработаны в Tecgraf в 1992—1993 годах для добавления некоторой гибкости в два отдельных проекта (оба были интерактивными графическими приложениями для конструкторских нужд в компании Petrobras). В SOL и DEL отсутствовали какие-либо управляющие конструкции, и Petrobras чувствовал растущую необходимость в добавлении к ним полноценного программирования.

Lua — это динамически типизированный язык, предназначенный для использования в качестве расширения или в качестве скриптового языка, и при этом достаточно компактный, чтобы поместиться на различных исполняющих платформах.

Компилятор — программа, выполняющая компиляцию. Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера). Входной информацией для компилятора (исходный код) является описание алгоритма или программа на объектно-ориентированном языке, а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

Компилировать — проводить трансляцию машинной программы с объектно-ориентированного языка на машинно-ориентированный язык.



Рис 1. Фазы компилятора.

В данной работе реализован front-end компилятора языка lua, включающий в себя стадии лексического, синтаксического и семантического анализа.

1. Лексический и синтаксический анализ

Лексический анализ — процесс аналитического разбора входной последовательности символов с целью получения на выходе последовательности символов, называемых «токенами». Группа символов входной последовательности, идентифицируемая на выходе процесса как токен, называется лексемой. В процессе лексического анализа производится распознавание и выделение лексем из входной последовательности символов.

Как правило, лексический анализ производится с точки зрения определённого формального языка или набора языков. Грамматика языка, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

В данной работе входной последовательностью является текстовый файл, представляющий собой программу, написанную на языке lua.

Цель такой конвертации состоит в том, чтобы подготовить входную последовательность для синтаксического анализатора, и избавить его от определения лексических подробностей в контекстно-свободной грамматике.

Синтаксический анализ — процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево).

Синтаксический анализатор — это программа или часть программы, выполняющая синтаксический анализ.

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное в виде абстрактного синтаксического дерева.



Рис 2. Абстрактное синтаксическое дерево для Алгоритма Евклида.

1.1. Реализация лексического и синтаксического анализаторов

В целях оптимизации и увеличения скорости разработки было принято решение воспользоваться программой ANTLR для создания лексического и синтаксического анализатора.

ANTLR (от англ. ANother Tool for Language Recognition — «ещё одно средство распознавания языков») — генератор нисходящих анализаторов для формальных языков. ANTLR преобразует контекстно-свободную грамматику в виде РБНФ в программу на C++, Java, C#, Python, Ruby. Используется для разработки компиляторов, интерпретаторов и трансляторов. На вход подаётся файл грамматики в формате ANTLR.

В данной работе генерируется лексический и синтаксический анализатор на языке Python.

Все инструкции в Lua выполнены в глобальной среде. Эта среда будет инициализирована обращением к `lua_open` и сохранится до обращения к `lua_close` или до завершения ведущей программы.

Глобальная среда может управляться Lua-кодом или ведущей программой, которая может читать и писать глобальные переменные, используя функции API из библиотеки, которая предоставлена Lua.

Глобальные переменные в Lua не должны быть объявлены. Любая переменная считается глобальной, пока не объявлена явно как локальная.

Модуль выполнения Lua назван составной частью. Это просто последовательность инструкций, которые выполнены последовательно. Каждая инструкция может факультативно сопровождаться точкой с запятой:

```
chunk ::= {stat [';']}
```

Lua представляет собой dynamically typed language. Переменные не имеют типов, а только значения. Следовательно, не имеется никаких определений типов на языке. Все значения несут их собственный тип. Помимо типа все значения также имеют тэг.

Имеются шесть базисных типов в Lua: `nil`, `number` (число), `string` (строка), `function` (функция), `userdata` (пользовательские данные) и `table` (таблица). `Nil` тип значения `nil`, чье основное свойство должно отличаться от любого другого значения. `Number` представляет реальные (двойная точность с плавающей запятой) числа, в то время как `string` имеет обычное значение. Lua нормально понимает 8-разрядные символы, так что строки могут содержать любой 8-разрядный символ, включая вложенные нули (`'\0'`).

Функции рассматриваются как значения первого класса (`first-class values`) в Lua. Функции могут быть сохранены в переменных, переданы как параметры другим функциям и возвращены как результаты.

1.1.1. Лексические соглашения

Идентификатором в Lua может быть любая строка символов, цифр и символов подчеркивания, не начинающаяся цифрой. Следующие слова зарезервированы, и не могут использоваться как идентификаторы:

```
and    break  do      else    elseif  return
end    for    function if      in      then
local  nil    not    or repeat until while
```

Lua представляет собой язык, чувствительный к регистру символов. Следующие строки обозначают другие токены:

```

~= <= >= < > == = + - *
( ) { } [ ] ; , . .. ... /

```

Литеральные строки могут быть разграничены одиночными или двойными кавычками, и могут содержать C-подобные управляющие последовательности: \a (bell), \b (backspace), \f (form feed), \n (newline), \r (carriage return), \t (horizontal tab), \v (vertical tab), \\ (backslash), \" (double quote), \' (single quote), и \newline. Символ в строке может также быть определен числовым значением, через управляющую последовательность \ddd, где ddd последовательность до трех десятичных цифр. Строки в Lua могут содержать любое 8-разрядное значение, включая вложенные нули, которые могут быть определены как \000.

Литеральные строки могут также быть разграничены парами [[...]].

Комментарии начинаются с двойного тире (--) и выполняются до конца строки.

Числовые константы могут быть написаны с факультативной целой частью и тоже факультативным дробной частями. Допустимо применение экспоненциальной формы записи. Примеры имеющих силу числовых констант:

```

3 3.0 3.1416 314.16e-2 0.31416E1

```

Для генерации лексического анализатора в входном файле для ANTLR используются следующие последовательности инструкций:

```

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"'\n]
                        | '\\' '\r'? '\n'
                        | DecimalEscape
                        | HexEscape
                        ;
fragment DecimalEscape : '\\' Digit
                      | '\\' Digit Digit
                      | '\\' [0-2] Digit Digit
                      ;
fragment HexEscape : '\\' 'x' HexDigit HexDigit;
fragment Digit : [0-9];

```

fragment HexDigit : [0-9a-fA-F];

COMMENT : '--[' NESTED_STR ']' -> skip;

LINE_COMMENT

```
: '--'  
(                                     // --  
| '[' '='*                           // --[==  
| '[' '='* ~('='|'['\"r'|\"n') ~('r'|\"n')* // --[==AA  
| ~('['\"r'|\"n') ~('r'|\"n')*           // --AAA  
) ('r'|\"n'|\"r'|\"n'|EOF)  
-> skip  
;
```

WS : [\t\u000C\r\n]+ -> skip;

SHEBANG : '# '!' ~('n'|\"r')* -> skip;

Полная грамматика языка Lua в БНФ:

chunk ::= {stat [';']} [laststat [';']]

block ::= chunk

stat ::= varlist '=' explist |
functioncall |
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 namelist ['=' explist]

laststat ::= return [explist] | break

funcname ::= Name { '.' Name } [':' Name]

varlist ::= var { ',' var }

var ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name

namelist ::= Name { ',' Name }

explist ::= {exp ',' } exp

exp ::= nil | false | true | Number | String | `...' | function |
prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | `(exp)'


```

functioncall ::= prefixexp args | prefixexp ':' Name args

args ::= '(' [explist] ')' | tableconstructor | String

function ::= function funcbody

funcbody ::= '(' [parlist] ')' block end

parlist ::= namelist ['`' ...'] | '`' ...

tableconstructor ::= '{' [fieldlist] '}'

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= '[' exp `]' `=' exp | Name `=' exp | exp

fieldsep ::= `,` | `;`

binop ::= `+` | `-` | `*` | `/` | `^` | `%` | `..` |
        `<` | `<=` | `>` | `>=` | `==` | `~=` |
        and | or

unop ::= `-` | not | `#`

```

1.1.2. Блоки

Блоком является список инструкций. Синтаксически блок равен составной части (chunk):

```
block ::= chunk
```

Блок может быть явно разграничен:

```
stat ::= do block end
```

Явные блоки полезны, чтобы управлять областью видимости (контекстом) локальных переменных.

В ANTLR это выражается в следующей форме:

```

chunk : block EOF;
block : stat* retstat?;
stat : 'do' block 'end'

```

1.1.3. Присваивания

Луа поддерживает многократные присваивания. Синтаксис определяет список переменных с левой стороны и список выражений с правой стороны. Элементы в обоих списках отделяются запятыми:

```

stat ::= varlist `=` explist
varlist ::= var {`,` var}
explist ::= exp {`,` exp}

```

Эта инструкция сначала оценивает все значения справа и возможные индексы слева, а затем делает присваивание. Так, код:

```
i = 3
i, a[i] = 4, 20
```

установит `a[3]` в 20. Многократное присваивание может использоваться, чтобы поменять местами два значения, например:

```
x, y = y, x
```

Два списка в многократном присваивании могут иметь различные длины. Перед собственно присваиванием, список значений будет откорректирован к длине списка имеющихся переменных.

Одиночное имя может обозначать глобальную переменную, локальную переменную или формальный параметр:

```
var ::= name
```

Квадратные скобки используются, чтобы индексировать таблицу:

```
var ::= prefixexp '[' exp ']'
varorfunc ::= var | functioncall
```

`varorfunc` должен иметь в качестве результата значение из таблицы, где поле, индексированное значением выражения `exp1`, получает назначенное ему значение.

Синтаксис `var.NAME` представляет собой только синтаксический аналог для выражения `var["NAME"]`:

```
var ::= prefixexp '.' Name
```

Значение присваиваний, оценок глобальных переменных и индексированных переменных может быть изменено методами тэгов. Фактически, назначение `x=val`, где `x` представляет собой глобальную переменную, является эквивалентным обращению `setglobal("x",val)`, а присваивание `t[i]=val` эквивалентно `settable_event(t,i,val)`.

Базисные выражения в Lua:

```
exp ::= prefixexp
exp ::= nil | false | true
exp ::= Number
exp ::= String
exp ::= function
exp ::= tableconstructor
exp ::= `...`
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | `(` exp `)`
```

В ANTLR это выражается в следующей форме:

```
stat : varlist '=' explist
```

```

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
    ;

varlist : var (' var)*;

var : (NAME | '(' exp ')') varSuffix varSuffix*;

varSuffix : nameAndArgs* (' exp ']' | '!' NAME);

prefixexp : varOrExp nameAndArgs*;

nameAndArgs : (':' NAME)? args;

args : '(' explist? ')' | tableconstructor | string;

```

1.1.4. Циклы for, while, repeat

Структуры управления if, while и repeat имеют обычное значение и синтаксис:

```

stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block} [else block] end

```

Выражение exp условия структуры управления может возвращать любое значение. Все значения, отличные от nil, рассматриваются как истина, только nil считается ложью.

Инструкция return используется, чтобы вернуть значения из функции или из chunk. Поскольку функции или составные части могут возвращать больше, чем одно значение, синтаксис для инструкции return:

```
stat ::= return [explist1]
```

Инструкция break может использоваться, чтобы завершить выполнение цикла, переходя к следующей инструкции сразу после цикла:

```
stat ::= break
```

break заканчивает самый внутренний вложенный цикл (while, repeat или for).

По синтаксическим причинам инструкции `return` и `break` могут быть написаны только как последние инструкции блока.

Инструкция `for` имеет две формы, по одной для чисел и таблиц. Числовая версия цикла `for` имеет следующий синтаксис:

```
stat ::= for Name '=' exp ',' exp '[' exp ] do block end
```

Таблица для инструкции `for` пересекает все пары (`index,value`) данной таблицы. Это имеет следующий синтаксис:

```
stat ::= for namelist in explist do block end  
namelist ::= Name { ',' Name }
```

В ANTLR это выражается в следующей форме:

```
stat : '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'  
    | 'break'  
    ;  
namelist : NAME (',' NAME)*;  
retstat : 'return' explist? ';' ;
```

1.1.5. Локальные объявления

Локальные переменные могут быть объявлены где-нибудь внутри блока. Объявление может включать начальное присваивание:

```
stat ::= local declist [init]  
declist ::= name { ',' name }  
init ::= '=' explist1
```

Если представлено начальное назначение, то оно имеет ту же самую семантику многократного назначения. Иначе все переменные инициализированы `nil`.

Область действия (контекст) локальных переменных начинается после объявления и продолжается до конца блока. Таким образом, код `local print=print` создает локальную переменную, названную `print`, чье начальное значение будет взято из глобальной переменной с тем же самым именем.

В ANTLR это выражается в следующей форме:

```
stat : 'local' 'function' NAME funcbody | 'local' namelist ('=' explist)? ;
```

1.1.6. Конструктор таблиц

Конструкторы таблиц представляют собой выражения, которые создают таблицы: каждый раз конструктор оценен, и новая таблица создана. Конструкторы могут использоваться, чтобы создать пустые таблицы или создать таблицу и инициализировать некоторые из полей (необязательно все). Общий синтаксис для конструкторов:

```
tableconstructor ::= '{' [fieldlist] '}'  
fieldlist ::= field { fieldsep field } [fieldsep]
```

```
field ::= '[' exp `]' `=' exp | Name `=' exp | exp
fieldsep ::= `,' | `;'
```

Выражения в списке назначены последовательным числовым индексам, начиная с 1. Использование выражений в стиле `[exp1] = exp2`, заносит в таблицу переменную `exp1` со значением `exp2`. Например такая запись:

```
a = {[f(k)] = g(y), x = 1, y = 3, [0] = b+c}
```

эквивалентна такому коду:

```
do
local temp = {}
temp[f(k)] = g(y)
temp.x = 1 -- or temp["x"] = 1
temp.y = 3 -- or temp["y"] = 3
temp[0] = b+c
a = temp
end
```

В ANTLR это выражается в следующей форме:

```
tableconstructor : '{' fieldlist? '}';
fieldlist field (fieldsep field)* fieldsep? ;
field : '[' exp `]' `=' exp | NAME `=' exp | exp;
fieldsep : `,' | `;'
```

1.1.7. Вызовы функций

Вызовы функций в Lua имеют синтаксис:

```
functioncall ::= prefixexp args
```

Сначала вычисляется `prefixexp`. Если значение имеет тип `function`, то эта функция будет вызвана с данными параметрами.

Форма:

```
functioncall ::= prefixexp `.` Name args
```

Может использоваться, чтобы вызвать `methods`. Обращение `v.name(...)` синтаксически аналогично `v.name(v, ...)`, за исключением того, что `v` будет включен в область видимости тела функции. Параметры имеют следующий синтаксис:

```
args ::= `(` [explist] `)`
args ::= tableconstructor
args ::= String
```

Все выражения параметра оценены перед обращением. Обращение в форме `f{...}` синтаксически аналогично `f({...})`, то есть список параметров представляет собой одиночную новую таблицу. Обращение в форме `f'...' (f'...' или f[...])` синтаксически аналогично `f('...')`, то есть список параметров представляет собой одиночную строку литералов.

Потому, что функция может возвращать любое число результатов, число результатов должно быть откорректировано прежде, чем они используются. Если функция вызвана как инструкция, то список возврата откорректирован к 0, таким образом отбрасывая все возвращенные значения. Если функция вызвана в месте, которое нуждается в одиночном значении (синтаксически обозначенном нетерминальным `expr1`), то список возврата откорректирован к 1, таким образом отбрасывая все возвращенные значения, но не первый. Если функция вызвана в месте, которое может использовать много значений (синтаксически обозначено нетерминальным `expr`), то никакая корректировка не будет сделана. Единственные места, которые могут обрабатывать много значений, это последние (или единственные) выражения в присваивании, в списке параметров или в инструкции `return`. Имеются примеры:

```
f()           -- 0 результатов
g(f(), x)     -- f() 1 результат
g(x, f())     -- g получает x и все значения, возвращенные f()
a,b,c = f(), x -- f() скорректирован к 1 результату (и c получает nil)
a,b,c = x, f() -- f() 2 результата
a,b,c = f()   -- f() 3 результата
return f()    -- возвращает все значения, возвращенные f()
return x,y,f() -- вернет a, b и все, что вернет f()
```

Синтаксис для определения функций такой:

```
function ::= function funcbody
funcbody ::= '(' [parlist] ')' block end
stat ::= function funcname funcbody
stat ::= local function Name funcbody
funcname ::= Name { '.' Name } [ ':' Name]
```

Инструкция

```
function f () ... end
```

является только синтаксическим аналогом для

```
f = function () ... end
```

а инструкция

```
function v.f () ... end
```

является синтаксическим аналогом для

```
v.f = function () ... end
```

Параметры действуют как локальные переменные, инициализированные со значениями параметра:

```
parlist ::= namelist [ ',' `...` ] | `...`
```

Инструкция

funcname ::= name ':' name

используется для определения методов, то есть функции, которые имеют неявный дополнительный параметр self .

Инструкция

function v:f (...) ... end

является только синтаксическим аналогом для

v.f = function (self, ...) ... end

В ANTLR это выражается в следующей форме:

```
stat : 'function' funcname funcbody | 'local' 'function' NAME funcbody;
funcname : NAME ('.' NAME)* (':' NAME)?;
funcbody : '(' parlist? ')' block 'end';
parlist : namelist (',' '...')? | '...';
```

1.2. Получение дерева разбора

По входящей грамматике ANTLR генерирует лексический и синтаксический анализатор. На вход классу LuaLexer подаётся входной файл. LuaLexer разбивает входной файл на токены, которые подаются в класс LuaParser, который генерирует AST. Полученное AST преобразуется в XML-формат для удобного представления и передачи для дальнейшего анализа.

Код	AST
local a b = "global" local function localFunc(x) return x end	<chunk> <block> <stat> <leaf>local</leaf> <namelist> <leaf>a</leaf> </namelist> </stat> <stat> <varlist> <var> <leaf>b</leaf> </var> </varlist> <leaf>=</leaf> <explist> <exp> <string> <leaf>"global"</leaf> </string> </exp> </explist> </stat> <stat>

	<pre> <leaf>local</leaf> <leaf>function</leaf> <leaf>localFunc</leaf> <funcbody> <leaf>(</leaf> <parlist> <namelist> <leaf>x</leaf> </namelist> </parlist> <leaf>)</leaf> <block> <retstat> <leaf>return</leaf> <explist> <exp> <prefixexp> <varOrExp> <var> <leaf>x</leaf> </var> </varOrExp> </prefixexp> </exp> </explist> </retstat> </block> <leaf>end</leaf> </funcbody> </stat> </block> <leaf>&lt;EOF&gt;</leaf> </chunk> </pre>
--	--

Таблица 1. Пример преобразования кода в AST.
После получения AST, совершается семантический анализ полученного дерева.

2. Семантический анализ

Семантический анализ – это проверка смысла, обращение к процедурам и функциям. Проверки, связанные с именами.

В данной работе производится обход дерева, полученного на этапе синтаксического анализа и производится:

- поиск переменных и функций в блоках, их запись в дерево;
- проверка на существование переменных и функций в выражениях в блоках;
- проверка на соответствие вызова функции сигнатуре функции;

Данные проверки и поиск переменных и функций воспроизводится только для переменных и функций, не объявленных в таблицах и не являющихся производным результатом функции или присваивания.

Семантический анализ производится методом поиска в глубину, с учётом стандартных функций и переменных языка:

Сигнатуры стандартных функций и переменных:

```
assert (v [, message])
collectgarbage ([opt [, arg]])
dofile ([filename])
error (message [, level])
_G
getfenv ([f])
getmetatable (object)
ipairs (t)
pcall (f, arg1, ...)
print (...)
rawequal (v1, v2)
rawget (table, index)
rawset (table, index, value)
select (index, ...)
setfenv (f, table)
setmetatable (table, metatable)
tonumber (e [, base])
tostring (e)
type (v)
unpack (list [, i [, j]])
_VERSION
xpcall (f, err)
module (name [, ...])
require (modname)
```

Стандартные модули:

```
coroutine
package
string
table
math
io
file:lines
os
debug
```

Для функций составляются словари, содержащие имя функции и минимальное количество переменных, необходимых этим функциям.

Алгоритм выполнения проверок в семантическом анализе:

```

keywords # Множество ключевых слов
defaultFunctions # Множество стандартных функций

globalVarTable = set() # Множество глобальных переменных
globalFuncTable = dict() # Словарь глобальных функций

semanticAnalyzer(root, topVarTable, bottomVarTable, topFuncTable):
    localVarTable = bottomVarTable
    localFuncTable = dict()
    localBottomVarTable = set()

    Для каждого child в root
        Если child.tag == "leaf" и не (child.text в keywords):
            Если root.tag == "var":
                unionFuncTable = localVarTable | topVarTable | topFuncTable |
defaultFunctions
                Если root.parent.tag == ("varlist" | "prefixexp"):
                    Если node(tag = "varSuffix") в root и не (child.text in
unionFuncTable)):
                        Ошибка("Использование необъявленной переменной")
                        globalVarTable <- child.text
                    Иначе Если root.parent.tag == "functioncall":
                        Если node(tag = "varSuffix") в root и не (child.text in
unionFuncTable)):
                            Ошибка("Использование необъявленной переменной")
                            Иначе если не (child.text в (unionFuncTable)):
                                Если не (unionFuncTable в topVarTable | localVarTable |
globalVarTable | defaultVariables)):
                                    Ошибка("Вызов необъявленной функции")
                                Иначе:
                                    count = calculateParams(root.parent.parent) # Подсчёт
параметров, переданных в функцию
                                    Если count < unionFuncTable[child.text]:
                                        Ошибка("Неверное количество аргументов,
переданных в функцию")
                                    Иначе:
                                        Если не (child.text in
(topVarTable | localVarTable | globalVarTable | defaultVariables |
unionFuncTable)):
                                            Ошибка("Использование необъявленной переменной")
                                        Иначе Если root.tag == "stat" и node(tag = "for") в root:
                                            myBottomVarTable <- child.text
                                        Иначе Если root.tag == "namelist":
                                            Если node(text = "for") в root.parent или root.parent.tag == "parlist":
                                                myBottomVarTable <- child.text # Передача параметров цикла или
функции в соответствующую область видимости
                                            Иначе:
                                                Если не (child.text в topVarTable) или (root.parent.tag == "stat" и
node(tag = "local") в root.parent):
                                                    localVarTable <- child.text
                                                Иначе Если root.tag == "funcname":
                                                    Если child.text == "self":
                                                        myBottomVarTable <- "self"
                                                        continue
                                                    Иначе:

```

```

                                globalFuncTable <- child.text
                                globalFuncTable[child.text] = calculateParams(root.parent.parent) # Подсчёт
параметров в сигнатуре функции
                                Иначе Если root.tag == "stat" и node(tag = "local") в root:
                                    Иначе:
                                        unionFuncTable = localFuncTable | topFuncTable
                                    Если child.tag == "block":
                                        semanticAnalyzer(child, localVarTable | topVarTable, myBottomVarTable,
unionFuncTable)
                                    Иначе:
                                        newVarTable, newFuncTable, newBottomVarTable = semanticAnalyzer(child,
localVarTable | topVarTable, set(), unionFuncTable)
                                        localVarTable = localVarTable | newVarTable
                                        myBottomVarTable = myBottomVarTable | newBottomVarTable
                                        localFuncTable = localFuncTable | newFuncTable

                                Если root.tag == "block":
                                    Если root.parend.tag == "chunk":
                                        updateVarTable(root, localVarTable | globalVarTable)
                                        updateFuncTable(root, localFuncTable | globalFuncTable)
                                    Иначе:
                                        updateVarTable(root, localVarTable)
                                        updateFuncTable(root, localFuncTable)
                                    Вернуть set(), dict(), set()

                                Вернуть localVarTable, localFuncTable, myBottomVarTable

```

Таким образом, процедура `semanticAnalyzer` возвращает обнаруженные внутри дерева переменные и функции и записывает их в `block`, а также передаёт в блоки функций и циклов переменные, объявленные в заголовке.

При этом производятся проверки на существование переменных и функций в выражениях в блоках и на соответствие вызова функции сигнатуре функции.

3. Тестирование

Для тестирования грамматики, лексического и синтаксического анализаторов, построенных ANTLR использовались стандартные тесты для lua версии 5.2.0. Тесты были успешно пройдены. Для запуска тестов можно воспользоваться следующей командой:

```
ls ../lua-5.2.0-tests | xargs -P 8 -I file -t python Lua.py ../lua-5.2.0-tests/file ../test-output/file
```

Для тестирования семантического анализатора были составлены дополнительные тесты:

Код	Результат
Поиск переменных и функций в блоках, их запись в дерево	
<pre>local a b = "global" function globalFunc1(x, y) local a, b, c = "local", "local", "local" d = "global" local function localFunc(x) return x end function globalFunc2(z) return z end function x.funcWithoutSelf(t) return t end function x.funcWithSelf(t) return t * self end for i = 1, 5 do print(i) end while a do print(a) end end</pre>	Ошибок не найдено
Проверка на существование переменных и функций в выражениях в блоках	
<pre>a = 10 b = a function globalFunc(x, y) local function localFunc(x) return x end</pre>	Ошибок не найдено

<pre> local c = 10 localFunc(c) localFunc(a) for x in b do print(x) end end </pre>	
<pre> a = 10 b = c b = a e.a = 10 function globalFunc(x, y) local function localFunc(x) return y end local c = 10 localFunc(d) for x in d do print(x) end end b = c globalFunc(a, b) localFunc(b) noFunc() </pre>	<p>Список ошибок:</p> <ul style="list-style-type: none"> Error with existance of variable: c Error with existance of variable: e Error with existance of variable: d Error with existance of variable: d Error with existance of variable: c Error with existance of function: localFunc Error with existance of function: noFunc
<p>Проверка на соответствие вызова функции сигнатуре функции</p>	
<pre> a = 10 b = a function globalFunc(x, y) local function localFunc(x) return x end local c = 10 localFunc(c) localFunc(a) localFunc(a).test() localFunc(a):test() for x in b do print(x) end end globalFunc(a, b) </pre>	<p>Ошибок не найдено</p>

<pre> f.a() g:a() g.a:a() function globalFunc(x, y) local function localFunc(x) return x end local a, b = "a", "b" localFunc(a) localFunc(a, b) localFunc() localFunc():test() localFunc().test() return x * y end local a, b, c = "a", "b", "c" globalFunc(a, b) globalFunc(a, b, c) globalFunc(a) </pre>	<p>Список ошибок:</p> <p>Error with existance of variable: f</p> <p>Error with existance of function: g</p> <p>Error with existance of variable: g</p> <p>Error With arguments arg: localFunc</p> <p>Error With arguments arg: localFunc</p> <p>Error With arguments arg: localFunc</p> <p>Error With arguments arg: globalFunc</p>
--	---

Таблица 2. Тестирование семантического анализатора.

4. Приложение

Для использования созданного приложения необходимо:

1. Установить ANTLR v4
2. Установить python версии 2.7
3. Запустить makefile, который генерирует лексический и синтаксический анализаторы
4. Использовать команду "python Lua.py [-debug] sample.lua > sample.xml"

Приложение обладает возможностью выводить отладочную информацию. При этом сохраняется возможность просмотреть AST. Иначе, при наличии ошибок, это будет невозможно.

Заключение

Дальнейшим развитием FrontEnd-а компилятора lua может быть улучшение семантического анализатора, путём расширения уже существующего функционала для таблиц. Также улучшение может быть проверка совпадения сигнатуры присваивания и возврата функций.

Также значительные улучшения можно внести написав собственный синтаксический анализатор, так как синтаксический анализатор, сгенерированный ANTLR, достаточно медленный и занимает до 95% времени работы программы.

Список используемой литературы

1. Википедия : свободная энцикл. – Электрон. дан. – [Б. м.], 2012. – URL: <http://ru.wikipedia.org/wiki/> (дата обращения: 02.06.2015).
2. Альфред Ахо, Рави Сети, Джеффри Ульман. Компиляторы. Принципы, технологии, инструменты. Издательство Вильямс, 2003. ISBN 5-8459-0189-8
3. Справочник по языку lua [электронный ресурс] URL: <http://www.lua.org/manual/5.1/manual.html> (дата обращения 02.06.2015)
4. Справочник по языку lua [электронный ресурс] URL: <http://www.lua.org/tests/5.2/> (дата обращения)