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

ОТЧЁТ

по технологической практике по конструированию и созданию компиляторов на тему:

«Компилятор языка Lua»

Студент _____ Матвеева Ю. А.

(Подпись, дата) (Фамилия И.О.)

Преподаватель _____ Ступников А. А.

(Подпись, дата) (Фамилия И.О.)

Москва, 2019

Реферат

Отчет по технологической практике по конструированию и созданию компиляторов 23 с., 5 ч., 5 рис., 3 источник, 1 приложение.

Компилятор, ANTLR, Lua, Python, ParrotVM.

Объектом разработки является компилятор языка Lua.

Цель работы – разработать приложение, выполняющее генерацию байт кода для исполнения на виртуальной машине ParrotVM на основе исходного кода языка Lua.

В задачи проекта входили:

- разработка технического задания на курсовой проект;
- проектирование системы;

В результате работы был создан компилятор языка Lua.

Введение

Цель работы — разработать приложение, выполняющее генерацию байткода для исполнения на виртуальной машине ParrotVM на основе исходного кода языка Lua.

Для достижения цели работы необходимо решить следующие задачи:

1. Разработать блок лексического и синтаксического анализа с явным построением дерева разбора для заданного исходного кода языка Lua;
2. Разработать блок семантического анализа и генерации кода для виртуальной машины ParrotVM с указанной точкой входа, соответствующей блоку chain в исходной программе.

1. Аналитический раздел

1.1. 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 выдает 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).

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

1.2. 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.

1.2.1. 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].

1.2.2. 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.

Coco/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.

1.3. 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.

1.4. 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.).

1.5. Семантический анализ

В процессе семантического анализа проверяется наличие семантических ошибок в программе, и 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.

1.5.1. 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;

4. Проверки, связанные с именами: к данной категории можно отнести проверку наличия в программе функции main, поскольку каждая C - программа должна содержать данную функцию как единственную точку входа.

Неуспешное завершение каждой из проверок означает семантическую ошибку.

1.6. Генерация кода

Генерация кода осуществляется с помощью нисхождения по абстрактному синтаксическому дереву, полученному в результате синтаксического анализа. Для каждого узла дерева определяется его тип и связанное с ним правило в грамматике языка. Исходя из этой информации генерируется байт-код для исполнения на виртуальной машине.

2. Конструкторский раздел

2.1. Структура компилятора

Структура разрабатываемого компилятора отражена на рисунке 1 в виде IDEF0 диаграммы.

Рис.1. Структура компилятора в нотации IDEF0.

2.2. Генерируемые классы

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

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

- LuaParser.py;
- LuaLexer.py;
- Lua.interp;
- LuaLexer.interp;
- LuaLexer.tokens;

- LuaListener.py;
- Lua.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.

2.3. Получение дерева разбора

Сгенерированный ANTLR синтаксический анализатор выдаёт абстрактное синтаксическое дерево в чистом виде, 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 точкой входа является нетерминал chunk, поэтому построение дерева следует начинать с вызова метода chunk() объекта класса синтаксического анализатора, являющейся корнем дерева.

2.4. Обнаружение и обработка лексических и синтаксических ошибок

Все ошибки, которые обнаруживаются лексическим и синтаксическим анализаторами ANTLR, по умолчанию выводятся в стандартный поток вывода ошибок. Данные ошибки возможно перехватить стандартным обработчиком ошибок языка на котором ведется разработка компилятора.

2.5. 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.g. При этом в методы передается текущий рассматриваемый узел с

информацией о родителе этого узла, а также о его потомках. Такой узел имеет следующие свойства:

- **treeNode.ChildCount** — количество детей у узла treeNode;
- **treeNode.GetChild(i)** — получение ребенка у узла treeNode под номером I;
- **treeNode.Type** — токен данного узла. Имеет тип int и должен сравниваться с константами, объявленными в классе лексера;
- **treeNode.Line** — номер линии токена в коде;
- **treeNode.CharPositionInLine** — позиция токена в коде от начала строки.

2.6. Генерация кода

Компилятор генерирует файл с расширением .pir (Parrot Intermediate Representation), являющийся промежуточном представлением генерируемого кода для виртуальной машины PirrotVM (Рис. 2).

```
.sub func

.param pmc a
.param pmc b

$P0 = new "Integer"
$P0 = a
$P1 = new "Integer"
$P1 = b
$P2 = new "Integer"
$P2 = $P0 * $P1
.return($P2)

.end
```

Рис. 2. Пример кода промежуточного представления для виртуальной машины ParrotVM

Данный файл содержит инструкции описывающие ход выполнения программы на языке высокого уровня, чем Ассемблер. Такой подход позволяет описывать код программы более естественным образом.

PIR позволяет зарегистрировать четыре типа:

- Integer - для целых чисел
- Floating point - для вещественных чисел
- String - для строковых данных
- Polymorphic container (PMC) - сложных типов (например массивы)

Ключевые слова начинаются с символа “.”. Примеры ключевых слов:

- sub - декларация функции.
- parm - объявления параметра функции
- end - конец блока видимости
- return - возврат значения из функции
- local - объявление локальной переменной в блоке

Для применения условий используются ключевое слово if. Для циклов используется unless. Переход к определенному блоку можно осуществить с помощью конструкции: “ goto label … label: “.

3. Использование компилятора

В составе общего решения был создан проект консольного приложения, написанного на языке Python, которое в качестве параметра принимает путь к файлу компилируемой программы, создаёт байт-файл в формате .pir.

3.1. Пример компиляции программы

Рассмотрим пример компиляции Lua программы. Текст программы приведён на рисунке 3.

```
function hello()
   a = 2
   print(a)
end

hello()
```

Рис 3. Пример программы на языке Lua.

В программе объявляется переменная a типа int, ей присваивается значение 2 и происходит вывод данного значения.

Воспользуемся стандартным плагином для среды PyCharm, ANTLR Tool и построим дерево разбора для данного исходного кода, руководствуясь грамматикой языка Ruby, заданной в формате ANTLR. Изображение дерева показано на рисунке 4.

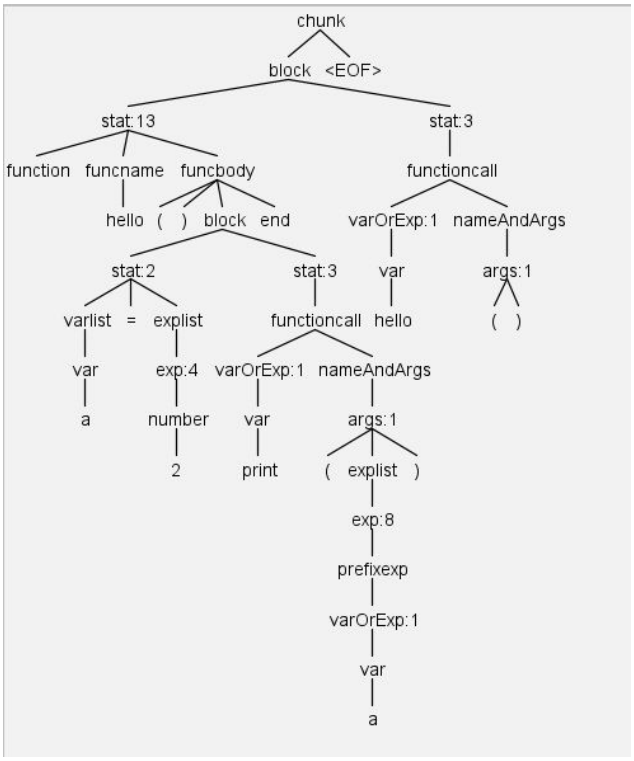

Рис 4. Пример дерева разбора

Разработанный компилятор на основе данного исходного Lua кода выполняет перевод в PIR язык. Приведенный выше код в синтаксисе PIR выглядит следующим образом рисунок 5.

```
.sub main

hello()

.end



.sub hello

$P0 = new "Integer"
$P0 = 2
say $P0

.end
```

Рис 5. Пример программы на PIR

Заключение

Рассмотрены основные фазы функционирования приложения, выполняющего компиляцию кода языка Lua в байт-код для ParrotVM.

Приведен обзор основных алгоритмов лексического и синтаксического анализа. Рассмотрены стандартные средства построения синтаксических анализаторов. Подробно описано использование пакета прикладных программ ANTLR4 для генерации исходного кода классов анализаторов по заданной грамматике языка Lua. Указаны возможные ошибки компиляции, обнаруживаемые в ходе лексического и синтаксического анализа.

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' namelist ('=' explist)?
   ;

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
   ;

/*
var
   : NAME | prefixexp '[' exp ']' | prefixexp '.' NAME
   ;

prefixexp
   : var | functioncall | '(' exp ')'
   ;

functioncall
   : prefixexp args | prefixexp ':' 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)
   ;
```