Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский политехнический университет Петра Великого

—

Институт кибербезопасности и защиты информации

**КУРСОВАЯ РАБОТА**

**~~ЮЛИ ВРЕДНУЛИ И~~ КРУТОГО МАКСИМА**

**Реализация грамматики языка программирования Lua**

по дисциплине «Формальные грамматики и теория компиляторов»

|  |  |  |
| --- | --- | --- |
| Выполнил  студент группы 4831001/80301 |  | Ю.С. Постникова |
|  | <подпись> |  |
| Выполнил  студент группы 4831001/80301 |  | М.А. Цюпко |
|  | <подпись> |  |
|  |  |  |
| Проверил  старший преподаватель | П.В. Семьянов | |
|  | <подпись> |  |

Санкт-Петербург

2021

Содержание

[1. ЯЗЫК ПРОГРАММИРОВАНИЯ LUA 3](#_Toc69903008)

[2. РЕАЛИЗОВАННЫЕ ГРАММАТИКИ 3](#_Toc69903009)

[2.1. Flex 3](#_Toc69903010)

[2.1.1. Идентификаторы 3](#_Toc69903011)

[2.1.2. Литеральные строки 4](#_Toc69903012)

[2.1.3. Числовая константа 5](#_Toc69903013)

[2.1.4. Комментарии 5](#_Toc69903014)

[2.1.5. Особенности регулярных выражений 5](#_Toc69903015)

[2.2. Выражения 6](#_Toc69903016)

[2.3. Арифметические операции 6](#_Toc69903017)

[2.4. Операции сравнения 6](#_Toc69903018)

[2.5. Конкатенация 6](#_Toc69903019)

[2.6. Приоритет операций 6](#_Toc69903020)

[2.7. Конструкторы таблиц 6](#_Toc69903021)

[2.8. Операторы присваивания 7](#_Toc69903022)

[2.9. Локальные переменные и блоки 7](#_Toc69903023)

[2.10. Управляющие конструкции 7](#_Toc69903024)

[3. УСТРАНЕНИЕ КОНФЛИКТОВ 7](#_Toc69903025)

[4. ВЫВОД 8](#_Toc69903026)

1. ЯЗЫК ПРОГРАММИРОВАНИЯ LUA

Lua ⸻ язык программирования расширений, разработан для поддержки общего процедурного программирования с возможностью описания данных. Lua также предлагает хорошую поддержку объектно-ориентированного, функционального и управляемого данными (data-driven) программирования. Lua предлагается как мощный и лёгкий встраиваемый скриптовый язык для любой программы, которая в этом нуждается.

Lua ⸻ динамически типизированный язык. Это означает, что значения не имеют типов; только значения. Язык не имеет определений типов. Все значения несут свой собственный тип.

Все значения в Lua первоклассные. Это означает, что все значения могут быть сохранены в переменных, переданы как аргументы другим функциям и возвращены как результаты.

В Lua существует восемь базовых типов: nil, boolean, number, string, function, userdata, thread и table.

1. РЕАЛИЗОВАННЫЕ ГРАММАТИКИ
   1. Flex и правила составления регулярных выражений

Данный файл представляет собой список регулярных выражений. Регулярные выражения составлены в соответствии с лексическими соглашениями языка Lua.

В главах ниже представлены описания нетривиальных регулярных выражений. Под тривиальными регулярными выражениями понимаются правила вида «один символ – один токен», например, арифметические выражения +, -, \*.

* + 1. Идентификаторы

*Имена* (также называемые идентификаторами) в Lua могут быть любой строкой из букв, цифр и подчеркиваний, не начинающейся с цифры. Идентификаторы используются для именования значений, полей таблиц и меток (labels).

Следующие *ключевые слова* зарезервированы и не могут использоваться как имена: and break do else elseif end false for function goto if in local nil not or repeat return then true until while.

* + 1. Литеральные строки

*Литеральные строки* могут быть ограничены сочетающимися одинарными или двойными кавычками, и могут содержать С-подобные управляющие последовательности: '\a' (bell), '\b' (backspace), '\f' (form feed), '\n' (newline), '\r' (carriage return), '\t' (horizontal tab), '\v' (vertical tab), '\\' (backslash), '\"' (двойная кавычка) и '\'' (апостроф [одинарная кавычка]). Обратный слеш, сопровождаемый реальным переходом на новую строку (newline), формирует переход строки (newline) в строке (string).

Строки в Lua могут содержать любое 8-битное значение, влючая встроенные нули, которые могут быть записаны как '\0'. Более того, возможно описать любой байт в литеральной строке его числовым значением. Это может быть сделано с помощью управляющей последовательности \xXX, где XX - это пара шестнадцатиричных цифр, или с помощью \ddd, где ddd - последовательность до трех десятичных цифр. (Обратите внимание, что если десятичная управляющая последовательность сопровождается цифрой, то она должна содержать ровно три цифры.)

Литеральные строки также могут быть определены с использованием длинных скобок. Мы определяем открывающую длинную скобку уровня n, как открывающую квадратную скобку, следующие за ней n знаков = и ещё одну открывающую квадратную скобку. Так, открывающая длинная скобка уровня 0 запишется так: [[, для уровня 1 - [=[, и так далее. Закрывающая длинная скобка определяется аналогично; например, закрывающая длинная скобка уровня 4 запишется так: ]====].

* + 1. Числовая константа

*Числовая константа* (или цифра) может быть записана с опциональной дробной частью и опциональной десятичной экспонентой, обозначенной буквой 'e' или 'E'. Lua также поддерживает шестнадцатеричные константы, которые начинаются с 0x или 0X. Шестнадцатеричные константы также допускают использование дробной части и бинарной экспоненты, обозначенной буквой 'p' или 'P'. Цифровая константа с разделительной точкой или экспонентой означает вещественное число; иначе она означает целое. Примеры допустимых целых чисел:

3 345 0xff 0xBEBADA

Примеры допустимых вещественных чисел:

3.0 3.1416 314.16e-2 0.31416E1 34e1

0x0.1E 0xA23p-4 0X1.921FB54442D18P+1

* + 1. Комментарии

Комментарии начинаются с двойного тире (--) в любом месте за пределами литеральной строки. Если текст, непосредственно следующий за --, не открывающая длинная скобка, то это короткий комментарий, который продолжается до конца строки. Иначе, это длинный комментарий, который продолжается до соответствующей закрывающей длинной скобки. Длинные комментарии часто используются для временного отключения кода.

* + 1. Ход работы и особенности регулярных выражений

На официальном сайте в описании регулярных строк не указано, что можно использовать символы [, ] внутри строки после знака экранирования. В ходе прохождения тестов это ошибка была устранена путем добавления \[ \] в EscapeSequence.

* 1. Bison и правила составления грамматики
     1. Функции

Lua, не смотря на свою простоту, как и большая часть современных языков поддерживает достаточно сложные конструкции. Например возможен следующий вариант вызова функции: *io.out.write:ln(‘hello’)*. Для возможности обработки такого вызова необходима рекурсия, позволяющая выражать два правила друг через друга. Bison поддерживает такое объявления правил, что существенно упрощает разработку грамматического анализатора с использованием данного средства. В результате чего, для добавления возможности анализа функций в языке Lua были разработаны следующие правила:

prefix\_exp: var

| function\_call

| '(' exp ')'

;

function\_call: prefix\_exp args

| prefix\_exp ':' NAME args

;

var: NAME

| prefix\_exp '[' exp ']'

| prefix\_exp '.' NAME

;

* + 1. Операции

Lua использует стандартный набор операций. В данной реализации синтаксического анализатора распознаются только два вида операций: бинарные и унарные. Выбор такого решения основывается на том, что разрабатываемый анализатор не должен распознавать контекст конкретной лексемы. Поэтому для обработки операций было использовано всего два правила:

exp: NIL

| exp BINOP exp

| exp MINUS exp

**…**

| MINUS exp

| UNOP exp

**…**

**;**

В языке Lua символы ‘<’ и ‘>’ используются не только в бинарный операциях, но и при объявлении атрибутов. В следствие чего, появилась необходимость в добавлении дополнительных правил:

exp: NIL

…

| exp '<' exp %prec MINUS

| exp '>' exp %prec MINUS

…

;

attr: '<' NAME '>'

| /\* empty \*/

;

1. УСТРАНЕНИЕ КОНФЛИКТОВ
   1. Циклические зависимости

Большинство правил языка Lua можно трактовать однозначно, но в процессе реализации грамматики языка, токен ‘exp’ может содержать в себе выражение вида NAME=NAME. Если разрешать эту проблему путем добавления в ‘exp’ правила ‘field’ помимо конфликтов shift/reduce возникало ошибка, нарушающая корректную работу анализатора и вызывающее циклическое преобразование ‘exp’ в ‘field’, в следствие чего анализатор зацикливался, и дальнейшая работа программы была невозможна. Это происходило из-за того, что правило ‘field’ выражалось через токен ‘exp’.

Проблема была решена при помощи уточнения контекста, т.е. теперь в ‘exp’ вместо правила ‘field’ используется правило ‘var’, в объявлении которого не используются циклические зависимости от ‘exp’.

* 1. Операции

В Lua символы ‘<’ и ‘>’ используется одновременно и в бинарных операциях, и при объявлении атрибутов. Поэтому для разрешения shift/reduce конфликтов необходимо было указать приоритет для правил, использующих эти символы:

exp '<' exp %prec MINUS

exp '>' exp %prec MINUS

* 1. Повторное объявление правил

При разработке языка большое количество shift/reduce конфликтов было устранено путем удаления правил, повторяющих друг друга.

1. Тестирование
2. Отладка

Для облегчения поиска правил, при описании которых были допущены ошибки использовались следующие методы отладки:

* Встроенные в lex и bison методы отладки, показывающие текущее состояние анализатора – считанный токен, обрабатываемое правило, стэк с состояниями. Включение макроса YYDEBUG\_VERBOSE позволяло добавить автогенерируемый вывод лексических ошибок;
* Для определения позиции символа, вызвавшего ошибку работы анализатора с точностью до строки, использовался модуль yyloc, чей функционал был расширен, чтобы получать информацию о столбце, в котором была вызвана ошибка.

1. ВЫВОД

В ходе выполнения данной курсовой работы были реализованы грамматики языка программирования Lua ver.5.4. Были изучены новые возможности Flex. Например, такое понятие, как сокращения. Можно задать регулярное выражение, назвать его и в правилах обращаться к нему по имени. Это особенно полезно, если регулярное выражение встречается в нескольких правилах. Bison и Flex существенно облегчают написание парсера языка программирования, однако данные средства не являются достаточными. Например, flex не поддерживает рекурсивные регулярные выражения, хотя подобные конструкции имеются в языке Lua.