Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский политехнический университет Петра Великого

—

Институт кибербезопасности и защиты информации

**КУРСОВАЯ РАБОТА**

**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

Оглавление

[1. ЯЗЫК ПРОГРАММИРОВАНИЯ LUA 3](#_Toc70417584)

[2. РЕАЛИЗОВАННЫЕ ГРАММАТИКИ 3](#_Toc70417585)

[2.1. Flex и правила составления регулярных выражений 3](#_Toc70417586)

[2.1.1. Идентификаторы 3](#_Toc70417587)

[2.1.2. Литеральные строки 4](#_Toc70417588)

[2.1.3. Числовая константа 5](#_Toc70417589)

[2.1.4. Комментарии 5](#_Toc70417590)

[2.2. Bison и правила составления грамматики 6](#_Toc70417591)

[2.2.1. Функции 6](#_Toc70417592)

[2.2.2. Операции 6](#_Toc70417593)

[3. УСТРАНЕНИЕ КОНФЛИКТОВ 7](#_Toc70417594)

[3.1. Циклические зависимости 7](#_Toc70417595)

[3.2. Операции 7](#_Toc70417596)

[3.3. Повторное объявление правил 8](#_Toc70417597)

[4. Тестирование 8](#_Toc70417598)

[5. ВЫВОД 9](#_Toc70417599)

1. ОСОБЕННОСТИ ЯЗЫКА 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.

Операции

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

Параллельное присваивание в Lua очень либерально, оно не требует точного соответствия числа переменных числу значений. Если значений больше, чем переменных, то последние значения отбрасываются, если значений меньше — последние переменные в списке получают значение nil.

x, y = y, x (параллельное присваивание)

В отличие от C и ряда других языков, даже числовой нуль (0) или пустая строка ("") с точки зрения логических операторов представляют значение «истина». Вычисление бинарных логических операторов традиционно выполняется до тех пор, пока результат не будет определён, причём в качестве результата возвращается последнее вычисленное подвыражение. Поэтому их можно использовать как тернарный оператор ?: в языке Си:

*-- аналогично r = ( a < b)? f(a) : f(b); в Си,*

r = ( a < b ) **and** f(a) **or** f(b)

*-- r получит значение функции f() от меньшего из двух значений: a и b,*

*-- при условии, что f(a) не равно nil или false.*

Таблицы

Таблица в Lua — это динамический гетерогенный ассоциативный массив, то есть множество пар «ключ-значение». Ключами могут быть значения любых типов Lua, кроме nil. Ключи также могут быть литералами (идентификаторами) Lua. Запись nil в элемент таблицы равносильна удалению данного элемента.

Таблицы являются единственным в Lua составным типом данных. Они являются фундаментом для всех пользовательских типов данных, таких как структуры, массивы, множества и другие:

empty = {} *-- Пустая таблица*

empty[1] = "первый" *-- Добавление элемента с целым индексом*

empty[3] = "второй" *-- Добавление элемента с целым индексом*

empty["третий"] = "третий" *-- Добавление элемента со строковым индексом*

empty[1] = **nil** *-- Удаление элемента из таблицы*

*-- Множество - индексы используются для хранения значений*

workDays = {["понедельник"]=**true**, ["вторник"]=**true**, ["среда"]=**true**, ["четверг"]=**true**, ["пятница"]=**true**}

workDays["суббота"] = **true** *-- Добавление субботы в число рабочих дней*

workDays["среда"] = **nil** *-- По средам больше не работаем*

Разделитель

Особенностью грамматика языка программирования языка программирования Lua является возможность опустить разделитель операторов ‘;’.

1. РЕАЛИЗОВАННАЯ ГРАММАТИКА
   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.

* + 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. 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

В первоначальной версии грамматики было 224 shift/reduce и 40 reduce/reduce конфликтов. Для их исправления грамматика была изменена следующим образом:

* 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’.

В результате исправления грамматики указанным выше образом количество конфликтов уменьшилось до 195 shift/reduce и 9 reduce/reduce.

Наибольшим приоритетом для исправления обладают reduce/reduce конфликты. Для уменьшения количества конфликтов данного вида были предприняты следующие меры:

* 1. Повторное использование токенов

Наибольшей сложностью в устранении обладают reduce/reduce конфликты, основная сложность заключается в том, что такие конфликты возникают при повторном использовании токенов без явного указания их контекста. Во многих языках программирования один и тот же токен может использоваться в нескольких правилах, например, NAME используется и как часть вызова функции, и как часть названия переменной. Такая ситуация вынуждает создавать дополнительные правила, которые уточняют контекст для проблемного токена. Основная проблема заключается в том, что подобные конструкции трудно распознать в. грамматике с первого взгляда. Для локализации конфликтов программа собиралась с флагом -v для создания файла с выводом всех состояний и перечнем конфликтов, которые они вызывают.

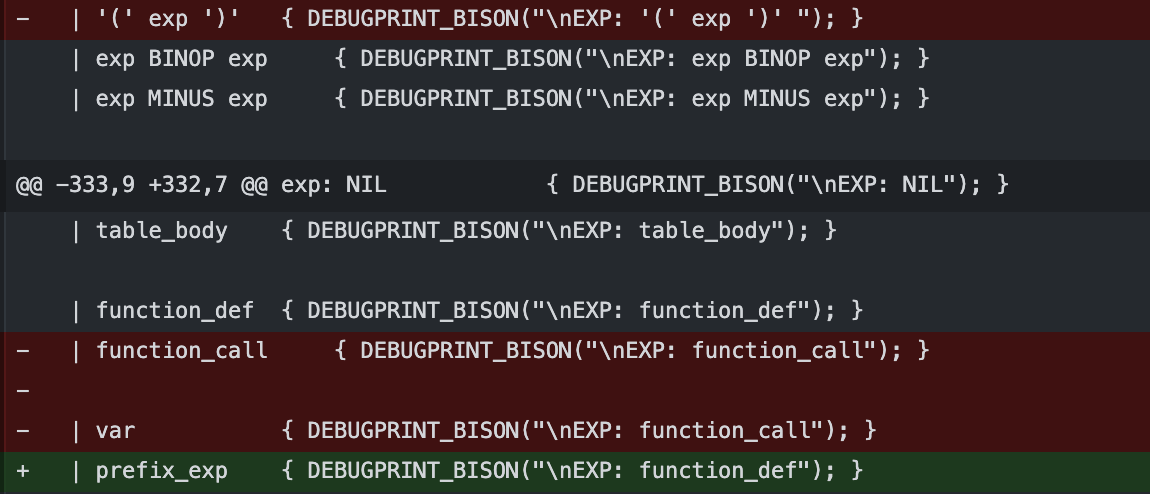


Рисунок 1 – правки для разработанной ранее анализатора

Как видно на рисунке 1, основные исправления связаны с правилом ‘exp’, в котором некорректно использовались ранее объявленные токены.

1. Разработка анализатора

При разработке анализатора основные сложности были связаны с реализацией синтаксических особенностей языка Lua:

1. Параллельное присваивание не вызвало особенных трудностей и было реализовано следующим набором правил:



Рисунок 2 – параллельное присваивание

1. Реализация таблиц также не вызвала трудностей, они были добавлены при помощи следующего правила:

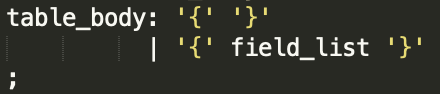


Рисунок 3 - таблицы

1. Для реализации особенности языка, которая позволяет опускать написание символа, завершающего строку, были разработаны следующие варианты правил:

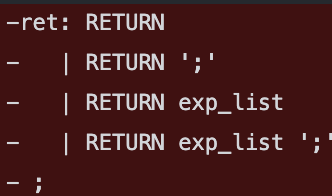


Рисунок 4 – все возможные варианты окончания команды

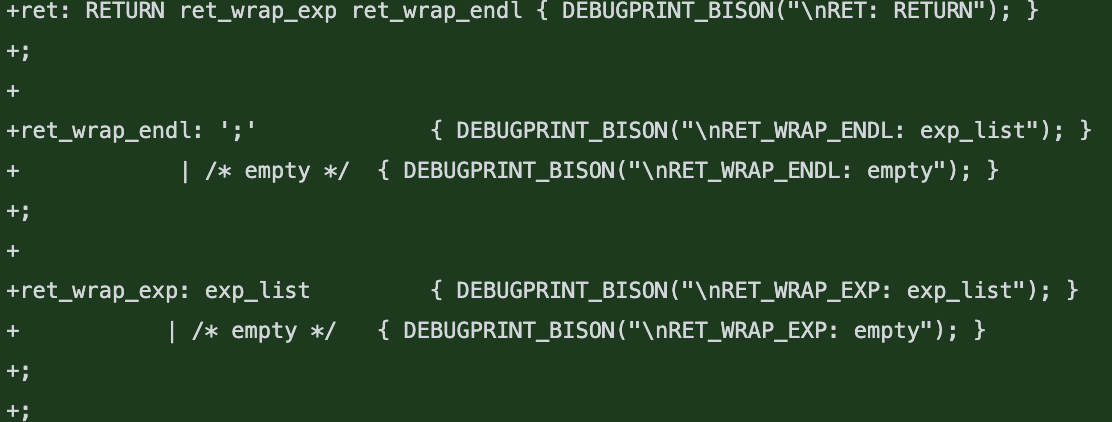


Рисунок 5 – использование особенностей bison

На рисунках 4-5 изображены два варианта реализации возможности опускать символ окончания строки. Второй вариант использует особенности bison.

1. Тестирование

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 добавило автогенерируемый вывод лексических ошибок;
* Для определения позиции символа, вызвавшего ошибку работы анализатора с точностью до столбца в строке, использовался модуль yyloc, чей функционал был расширен, чтобы получать информацию не только о строке, но и о столбце (рисунок 1);
* Вывод дополнительной информации о работе анализатора, содержащей в себе порядок применённых им правил (рисунок 2).

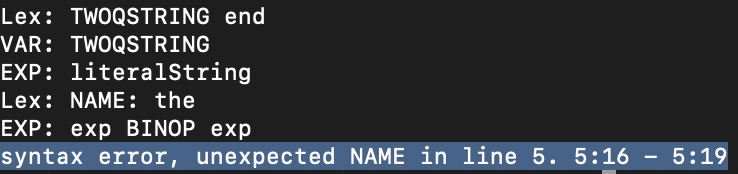


Рисунок 6 – вывод строки и столбца модулем yyloc

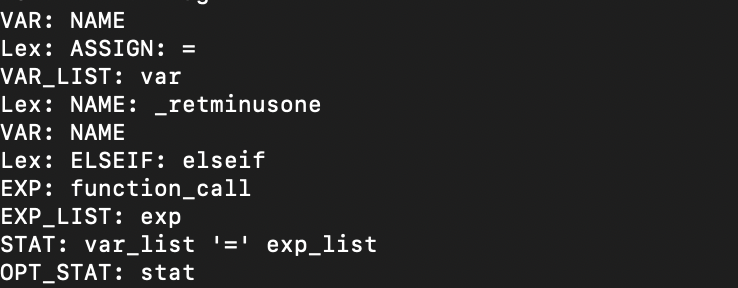


Рисунок 7 – пример вывода дополнительной отладочной информации

В процессе выполнения работы наибольшую эффективность при отладке показало совместное использование yyloc и вывода дополнительной отладочной информации. Такой способ отладки позволял за наименьший промежуток времени локализовать ошибку и внести изменения в анализатор языка.

В качестве тестов были использованы официальные тесты с сайта Lua и некоторые проекты с github.

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 используются в языке Lua.

Bison может выступать в качестве средства для разработки полноценного компилятора языка любой сложности, генерируемые им LR(1) таблицы не накладывают почти никаких ограничений. Основные сложности возникают из-за предоставляемых bison’ом средств объявления грамматики языка, эти средства разрабатывались под влиянием языков не поддерживающих ООП, из-за чего разработка компилятора для современного языка является достаточно сложным процессом.

В случае разработки синтаксического анализатора, bison достаточно удобен, т.к. предоставляет удобный, хотя и достаточно низкоуровневый, интерфейс для разработки грамматики, поддерживающий достаточно эффективный механизм разрешения конфликтов.