Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Расчетно-пояснительная записка

По курсу конструирование компиляторов

НА ТЕМУ:

Фронтенд компилятора языка LUA

Студент ИУ7-22М _____ **Н.В.Уточкина**
(Группа) (Подпись, дата) (И.О.Фамилия)

Преподаватель _____ **А.А. Ступников**
(Подпись, дата) (И.О.Фамилия)

2020 г.

Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

ЗАДАНИЕ
по курсу конструирование компиляторов

Студент группы ИУ7-22М Уточкина Наталия Витальевна
(фамилия, имя, отчество)

Цель: овладение алгоритмами синтаксического анализа текста и навыками построения грамматик для разбора и аналитики входного текста.

Задачи:

- закрепление на практике имеющихся и приобретение новых специализированных знаний, умений, навыков и компетенций по конструированию компиляторов;
- анализ существующих проблем в выбранной предметной области и определение возможных путей для их решения;
- анализ существующего программного обеспечения (ПО) предметной области. Обзор методов, алгоритмов и программного обеспечения, которые могут быть использованы для решения выявленных проблем предметной области и разработки нового ПО;

Оглавление

Введение

Основания для разработки

Разработка ведется в рамках проведения учебной практики по конструированию компиляторов на кафедре «Программное обеспечение ЭВМ и информационные технологии» факультета «Информатика и системы управления» МГТУ им. Н. Э. Баумана.

Назначение разработки

Построение фронтенда компилятора языка LUA, реализующего разбор текста на языке LUA и построение описывающего его типизированного абстрактного синтаксического дерева.

Предоставить возможность вывода дерева в файл в формате json.

Существующие аналоги

Lua - это легкий, высокоуровневый, многопарадигменный язык программирования, предназначенный в первую очередь для встроенного использования в приложениях. Lua является кроссплатформенным, так как интерпретатор скомпилированного байт-кода написан на языке ANSI C, и Lua имеет относительно простой C API для встраивания его в приложения.

Lua был первоначально разработан в 1993 году как язык для расширения программных приложений, чтобы удовлетворить растущий спрос на кастомизацию в то время.

Оригинальный интерпретатор языка написан на языке C и свободно распространяется, также присутствуют реализации синтаксических анализаторов на различных языках программирования на таких ресурсах как GitHub.

Описание системы

Проект должен представлять собой фронтенд компилятора, написанный на языке TypeScript, который принимает на вход программу на языке LUA и строит его типизированное синтаксическое дерево, затем выводит его в формате json.

Общие требования к системе

1. Возможность вывода синтаксического дерева в json.
2. Обработка ошибок, исключения и ошибки не приводят к зависанию или падению процесса.

Аналитический раздел

Язык программирования — формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы и действия, которые выполнит компьютер под ее управлением. Различают языки низкого уровня, к примеру, язык ассемблера, и языки высокого уровня, к которым относится большинство современных, включая LUA.

Для того, чтобы язык можно было применить, то есть записывать на нем программы, а в последствии их исполнять, язык необходимо реализовать.

Реализацией языка программирования называют создание комплекса программ, обеспечивающих работу на этом языке. Такой набор программ называется системой программирования. Основу каждой системы программирования составляет транслятор. Это программа, переводящая текст на языке программирования в форму, пригодную для исполнения (на другой язык). Такой формой обычно являются машинные команды, которые могут непосредственно исполняться компьютером. Совокупность машинных команд данного компьютера (процессора) образует его систему команд или машинный язык. Программу, которую обрабатывает транслятор, называют исходной программой, а язык, на котором записывается исходная программа —входным языком этого транслятора.

Фронтенд компилятора

Фронтенд компилятора — первый из двух ключевых компонентов компилятора, о котором можно сказать следующее:

1. Frontend получает на вход текст программы.

2. Если в тексте есть синтаксические и семантические ошибки, Frontend находит их.
3. Если текст корректен, фронтенд строит абстрактное синтаксическое дерево (AST), хранящее логическую модель файла исходного кода.
4. Превращение текста в AST происходит в три этапа:
    a. Лексический анализ (разделение текста на токены);
    b. Синтаксический анализ (токены собираются по грамматике в абстрактное синтаксическое дерево AST);
    c. Семантический анализ (постобработка AST).

Компилятор, интерпретатор, конвертор

Компилятор, обрабатывая исходную программу, создает эквивалентную программу на машинном языке, которая называется также объектной программой или объектным кодом. Объектный код, как правило, записывается в файл, но не обязательно представляет собой готовую к исполнению программу. Для программ, состоящих из многих модулей, может образовываться много объектных файлов. Объектные файлы объединяются в исполняемый модуль с помощью специальной программы-компоновщика, которая входит в состав системы программирования. Возможен также вариант, когда модули не объединяются заранее в единую программу, а загружаются в память при выполнении программы по мере необходимости. Интерпретатор, распознавая, как и компилятор, исходную программу, не формирует машинный код в явном виде. Для каждой операции, которая может потребоваться при исполнении исходной программы, в программе-интерпретаторе заранее заготовлена машинная команда или команды. «Узнав» очередную операцию в исходной программе, интерпретатор выполняет соответствующие команды, потом —

следующие, и так всю программу. Интерпретатор — это переводчик и исполнитель исходной программы. Различие между интерпретаторами и компиляторами можно проиллюстрировать такой аналогией. Чтобы пользоваться каким-либо англоязычным документом, мы можем поступить по-разному. Можно один раз перевести документ на русский, записать этот перевод и потом пользоваться им сколько угодно раз. Это ситуация, аналогичная компиляции. Заметьте, что после выполнения перевода переводчик (компилятор) больше не нужен. Интерпретация же подобна чтению и переводу «с листа». Перевод не записывают, но всякий раз, когда нужно воспользоваться документом, его переводят заново, каждый раз при этом используя переводчик.

Рис. 1.1.Схема работы компилятора и интерпретатора

Нетрудно понять преимущества и недостатки компиляторов и интерпретаторов. Компилятор обеспечивает получение быстрой программы на машинном языке, время работы которой намного меньше времени, которое будет затрачено на выполнение той же программы интерпретатором. Однако компиляция, являясь отдельным этапом

обработки программы, может потребовать заметного времени, снижая оперативность работы. Скомпилированная программа представляет собой самостоятельный продукт, для использования которого компилятор не требуется. Программа в машинном коде, полученная компиляцией, лучше защищена от внесения несанкционированных искажений, раскрытия лежащих в ее основе алгоритмов.

Интерпретатор исполняет программу медленно, поскольку должен распознавать конструкции исходной программы каждый раз, когда они исполняются. Если какие-то действия расположены в цикле, то распознавание одних и тех же конструкций происходит многократно. Зато интерпретатор может начать исполнение программы сразу же, не затрачивая времени на компиляцию, — получается оперативней. Интерпретатор, как правило, проще компилятора, но его присутствие требуется при каждом запуске программы. Поскольку интерпретатор исполняет программу по ее исходному тексту, программа оказывается незащищенной от постороннего вмешательства. Но даже при использовании компилятора получающаяся машинная программа работает, как правило, медленнее и занимает в памяти больше места, чем такая же программа, написанная вручную на машинном языке или языке ассемблера квалифицированным программистом. Компилятор использует набор шаблонных приемов для преобразования программы в машинный код, а программист может действовать нешаблонно, отыскивая оптимальные решения. Разработчики тратят немало усилий, стремясь улучшить качество машинного кода, порождаемого компиляторами.

В действительности различие между интерпретаторами и компиляторами может быть не столь явным. Некоторые интерпретаторы выполняют предварительную трансляцию исходной программы в промежуточную форму, удобную для последующей интерпретации. Некоторые компиляторы могут не создавать файла объектного кода.

Например, Turbo Pascal может компилировать программу в память, не записывая машинный код в файл, и тут же ее запускать. Поскольку компилирует Turbo Pascal быстро, то такое его поведение вполне соответствует работе интерпретатора.

Существуют трансляторы, переводящие программу не в машинный код, а на другой язык программирования. Такие трансляторы иногда называют конверторами. Например, в качестве первого шага при реализации нового языка часто разрабатывают конвертор этого языка в язык Си. Дело в том, что Си — один из самых распространенных и хорошо стандартизованных языков. Обычно ориентируются на версию ANSI Си — стандарт языка, принятый Американским Национальным Институтом Стандартов (American National Standards Institute, ANSI). Компиляторы ANSI Си есть практически в любой системе.

Кросскомпиляторы (cross-compilers) генерируют код для машины, отличной от той, на которой они работают.

Пошаговые компиляторы(incremental compilers) воспринимают исходный текст программы в виде последовательности задаваемых пользователем шагов (шагом может быть описание, группа описаний, оператор, группа операторов, заголовок процедуры и др.); допускается ввод, модификация и компиляция программы по шагам, а также отладка программ в терминах шагов.

Динамические компиляторы (Just-in-Time — JIT compilers), получившие в последнее время широкое распространение, транслируют промежуточное представление программы в объектный код во время исполнения программы.

Двоичные компиляторы(binary compilers) переводят объектный код одной машины (платформы) в объектный код другой. Такая разновидность компиляторов используется при разработке новых аппаратных архитектур, для переноса больших системных и прикладных

программ на новые платформы, в том числе —операционных систем, графических библиотек и др.

Транслятор — это большая и сложная программа. Размер программы-транслятора составляет от нескольких тысяч до сотен тысяч строк исходного кода. Вместе с тем разработка транслятора для не слишком сложного языка — задача вполне посильная для одного человека или небольшого коллектива.

Конструкторский раздел

Описываемая программа состоит из двух основных частей: лексического и синтаксического анализаторов.

Лексический анализатор

Лексический анализатор производит токенизацию - преобразует поток входных символов в поток типизированных объектов-токенов.

Токены описываются с помощью имени - строки в верхнем регистре и регулярного выражения, соответствующего ему. При этом важен порядок определения токенов, т.к. некоторые регулярные выражения описывают подмножество строк, полностью входящее в подмножество, описываемое другим регулярным выражением. Например, если сначала описать знак присваивания “=”, а затем сравнения “==”, то второй никогда не будет получен в результате работы лексического анализатора.

Токенами являются:

1. Имена ([a-zA-Z_][a-zA-Z_0-9]*).
2. Строки ((\'.*\')|(\".*\")).
3. Числа ((\d*\.\d+)|(\d+\.\d*)|(\d+)).
4. Зарезервированные слова (к примеру “return” или “while”).
5. Символы операций (к примеру “+” или “#”).

Лексический анализатор при выполнении правила меняет переменную yytext - строка, которая совпала с определенным токеном (лексема). Каждое правило при этом возвращает токен. Таким образом строка

```
result = (a + b)*12+(c/3)
```

будет иметь следующий набор записей “лексема” - “токен”:

```
[
```

```
{
    "lexeme": "result",
    "token": "NAME"
},
{
    "lexeme": "=",
    "token": "ASSIGNMENT"
},
{
    "lexeme": "(",
    "token": "ROUND_LBRACKET"
},
{
    "lexeme": "a",
    "token": "NAME"
},
{
    "lexeme": "+",
    "token": "PLUS"
},
{
    "lexeme": "b",
    "token": "NAME"
},
{
    "lexeme": ")",
    "token": "ROUND_RBRACKET"
},
{
    "lexeme": "*",
    "token": "TIMES"
},
{
    "lexeme": 12,
    "token": "NUMBER"
},
{
    "lexeme": "+",
    "token": "PLUS"
},
```

```
    {
        "lexeme": "(",
        "token": "ROUND_LBRACKET"
    },
    {
        "lexeme": "c",
        "token": "NAME"
    },
    {
        "lexeme": "/",
        "token": "DIVIDE"
    },
    {
        "lexeme": 3,
        "token": "NUMBER"
    },
    {
        "lexeme": ")",
        "token": "ROUND_RBRACKET"
    }
]
```

Синтаксический анализатор

Синтаксический анализатор производит чтение потока токенов и с помощью правил приведения строит в памяти синтаксическое дерево, описывающее входной текст в терминах языка программирования.

Поток токенов синтаксическому анализатору подается в результате работы лексического анализатора. Затем он с помощью SLR-анализатора преобразует наборы токенов лексического анализатора в вершины синтаксического дерева.

Вершины синтаксического дерева описываются в виде классов. В качестве первой вершины всегда выступает класс Chunk.

Каждое правило лексического анализатора задается парой из структуры, описывающей вершину синтаксического дерева и самого правила вывода.

Например, выражение описывается следующим образом:

BNF :

```
"expression": [
   ["NIL", "$$ = new yy.Expression($1)"],
   ["FALSE", "$$ = new yy.Expression($1)"],
   ["TRUE", "$$ = new yy.Expression($1)"],
   ["number", "$$ = new yy.Expression($1)"],
   ["literal_string", "$$ = new yy.Expression($1)"],
   ["VARARG", "$$ = new yy.Expression($1)"],
   ["funcdef", "$$ = new yy.Expression($1)"],
   ["table", "$$ = new yy.Expression($1)"],
   ["binop", "$$ = new yy.Expression($1)"],
   ["unop", "$$ = new yy.Expression($1)"],
   ["prefixexp", "$$ = new yy.Expression($1)"]
]
```

Класс Expression с описанным типом:

```
type ExpressionType = Number |
   LiteralString |
   FunctionDef |
   Table |
   BinaryOp |
   UnaryOp |
   PrefixExpression |
   "VARARG" | "FALSE" | "TRUE";

export class Expression {
   constructor(private expression:ExpressionType) {
       logger.parserLog("Expression:", [expression]);
       if (expression instanceof Number) {
           this.type = "Number";
           return;
       }
```

```
            if (expression instanceof LiteralString) {
                this.type = "LiteralString";
                return;
            }
            if (expression instanceof FunctionDef) {
                this.type = "FunctionDef";
                return;
            }
            if (expression instanceof TableConstructor) {
                this.type = "TableConstructor";
                return;
            }
            if (expression instanceof BinaryOp) {
                this.type = "BinaryOp";
                return;
            }
            if (expression instanceof UnaryOp) {
                this.type = "UnaryOp";
                return;
            }
            if (expression instanceof PrefixExpression) {
                this.type = "PrefixExpression";
                return;
            }
            this.type = expression;
        }
        private type:string = "";
    }
```

В классе запоминается само выражение и его тип в виде строки.

Пользовательский интерфейс

Программа выполняется в браузере и обладает следующим интерфейсом:

```
function insert (index, value)
  if not statetab[index] then
    statetab[index] = {n=0}
  end
  table.insert(statetab[index], value)
end

local N  = 2
local MAXGEN = 10000
local NOWORD = "\n"

statetab = {}
local w1, w2 = NOWORD, NOWORD
for w in allwords() do
  insert(prefix(w1, w2), w)
  w1 = w2; w2 = w;
end
insert(prefix(w1, w2), NOWORD)

w1 = NOWORD; w2 = NOWORD
for i=1,MAXGEN do
  local list = statetab[prefix(w1, w2)]
  local r = math.random(table.getn(list))
  local nextword = list[r]
  if nextword == NOWORD then return end
  io.write(nextword, " ")
  w1 = w2; w2 = nextword
end
```

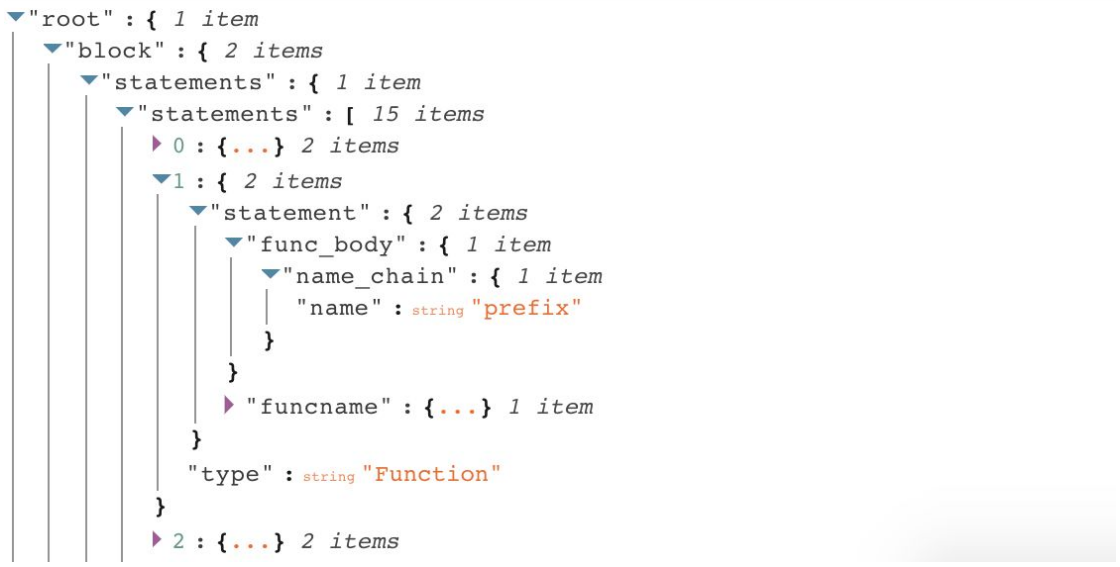

Рисунок 1. Вид программы

На странице есть поле для ввода кода. В него можно вставить код, можно его редактировать. По нажатию кнопки Enter строится AST,

которое располагается ниже на странице. В отображенном AST можно открывать и раскрывать ноды для удобного просмотра.

Технологический раздел

Выбор средств разработки

Выбор целевой платформы

Программный продукт мультиплатформенный. Он написан на языке TypeScript. Программу, написанную на языке TypeScript можно использовать на любой платформе, где есть браузер.

Выбор языка программирования

Для написания программы выбран язык TypeScript, т.к. это одно из условий курсового проекта. Кроме того язык удобен для выполнения учебных проектов.

Выбор среды разработки и отладки

В качестве среды для разработки использовался редактор WebStorm. Редактор имеет множество настроек, позволяет управлять системой контроля версий, сохранять и выполнять скрипты для запуска программы.

Система контроля версий

В процессе разработки программы использовалась система контроля версий Git. Система контроля версий позволяет вносить в проект атомарные изменения, направленные на решения каких-либо задач. В случае обнаружения ошибок или изменения требований, внесенные изменения можно отменить. Кроме того, с помощью системы контроля версий решается вопрос резервного копирования.

Особенности Git:

1. Предоставляет широкие возможности для управления изменениями проекта и просмотра истории изменений;
2. Данная система контроля версий является децентрализованной, что позволяет иметь несколько независимых резервных копий проекта;
3. Поддерживается хостингами репозиториев GitHub, GitLab и BitBucket.

Выбор средств конструирования синтаксического анализатора

Для лексического анализа использовалась библиотека Lexer. Для синтаксического анализа была выбрана библиотека Jison. Обе библиотеки написаны на чистом JavaScript, то есть не имеют внешних зависимостей. Их можно легко использовать при написании программы на языке TypeScript, т.к. язык компилируется в JavaScript.

Функциональность Jison совпадает с классическими UNIX-утилитами Flex/Bison. Отсутствие функций, вроде автоматического распознавания синтаксиса языка в БНФ позволяет лучше понять принцип работы лексического и синтаксического разборов, лежащих в основе конструктора синтаксического анализатора. Вместо лексического анализатора встроенного в Jison использовался упомянутый ранее Lexer, т.к. позволяет более комфортно описывать правила.

Известны такие аналоги Jison как ANTLR, APG, Neary, Canopy, которые тоже позволяют использовать их в программе на TypeScript.

Список использованной литературы

1. Конструирование компиляторов. Учебное пособие. Автор Сергей Свердлов. [Электронный ресурс]. Режим доступа - http://www.uni-vologda.ac.ru/~c3c/CompilersConstruction/Sergey_Sverdlov_Compiler_Construction_(electronic%20version).pdf, свободный (Дата обращения: 12.09.2020)
2. Jison.[Электронный ресурс]. Режим доступа - https://zaa.ch/jison/docs/, свободный (Дата обращения: 14.09.2020).
3. Lexer.[Электронный ресурс]. Режим доступа - https://github.com/aaditmshah/lexer, свободный (Дата обращения: 14.09.2020)
4. Bison Guide.[Электронный ресурс]. Режим доступа - http://dinosaur.compilertools.net/bison/bison_4.html#SEC7 , свободный (Дата обращения: 14.09.2020)