|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **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 **КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ (ИУ6)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

**ОТЧЕТ**

|  |  |
| --- | --- |
| **По домашнему заданию №2** |  |

**Название:** Лексические и синтаксические анализаторы

**Дисциплина:** Машинно-зависимые языки и основы компиляции

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ИУ-42б |  |  | С.В. Астахов |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  |  |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

Москва, 2021

Задание

Разработать грамматику и распознаватель описаний записей с вариантами языка программирования Pascal. Предусмотреть следущие типы элементов: Real, Integer, Char, Byte.

Например: Var se:record I,k: byte; case of 1:(h:integer); 2:(ch:char) end;

Цель

Закрепление знаний теоретических основ и основных методов приемов разработки лексических и синтаксических анализаторов регулярных и контекстно-свободных формальных языков.

Описание грамматики

*В форме Бэкуса-Наура:*

<запись> ::= var <идентификатор>:record <постоянные поля>; <поля с вариантами> end;

|var <идентификатор>:record <постоянные поля> end;

<постоянные поля> ::= <поля одного типа>

| <поля одного типа>;<постоянные поля>

<поля с вариантами> ::= case <идентификатор> of <случаи>

<случаи> = <случай> | <случай>;<случаи>

<случай> ::= <ключ>: (<поля одного типа>)

<поля одного типа> ::= <идентификаторы>: <тип>

<идентификаторы> ::= <идентификатор> | <идентификатор>,<идентификаторы>

<идентификатор> ::= <буква> | <буква><идентификатор> | <цифра><идентификатор>

<ключ> ::= <цифра> | <цифра><ключ>

<тип> ::= byte | char | integer | real

<цифра> ::= 0|1|2|3|4|5|6|7|8|9

<буква> ::= \_ | a | b | … | z

*В форме синтаксических диаграмм:*

Соответствующие конструкциям грамматики диаграммы представлены на рисунках 1-8:

Рисунок 1 — синтаксическая диаграмма «запись»

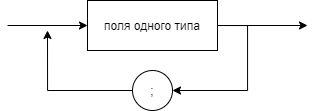
Рисунок 2 — синтаксическая диаграмма «постоянные поля»

Рисунок 3 — синтаксическая диаграмма «поля с вариантами»

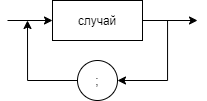
Рисунок 4 — синтаксическая диаграмма «случаи»

Рисунок 5 — синтаксическая диаграмма «случай»

Рисунок 6 — синтаксическая диаграмма «поля одного типа»

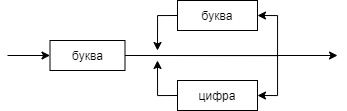
Рисунок 7 — синтаксическая диаграмма «идентификаторы»

Рисунок 8 — синтаксическая диаграмма «идентификатор»

Данная грамматика является LL(k) грамматикой, поэтому будем использовать метод рекурсивного спуска.

Код программы

function validID(x) {

x = x.trim()

console.log("validingID:" + x)

if (/[\_a-z]$/.test(x[0])) {

while ((/[\_a-z0-9]$/.test(x[0])) && x.length > 0) {

x = x.slice(1)

}

} else {

throw "Wrong ID"

}

x = x.trim()

return x

}

function validIDs(x) {

x = x.trim()

console.log("validingIDs:" + x)

x = validID(x)

if (x[0] == ",") {

x = x.slice(1)

x = validIDs(x)

} else {

console.log("validingIDs end")

}

return x

}

function validType(x) {

x = x.trim()

console.log("validingType:" + x)

let xcpy = x

if (x.slice(0, 4) == "byte" || x.slice(0, 4) == "char" || x.slice(0, 4) == "real") {

x = x.slice(4)

} else if (x.slice(0, 7) == "integer") {

x = x.slice(7)

} else {

throw "Wrong Type"

}

logOnUi("Type: "+xcpy.slice(0,xcpy.length-x.length))

x = x.trim()

return x

}

function validSameFields(x) {

x = x.trim()

console.log("validingSameFields:" + x)

let xcpy

xcpy = x

x = validIDs(x)

logOnUi("\nFields: "+ xcpy.slice(0,xcpy.length-x.length))

x = x.trim()

if (x[0] == ":") {

x = x.slice(1)

x = x.trim()

x = validType(x)

x = x.trim()

} else {

throw "Wrong Same Fields"

}

return x

}

function validKey(x) {

x = x.trim()

console.log("validingKey:" + x)

if (/[0-9]$/.test(x[0])) {

while ((/[0-9]$/.test(x[0])) && x.length > 0) {

x = x.slice(1)

}

} else {

throw "Wrong Key"

}

x = x.trim()

if (x[0] != ":") {

throw "Wrong Key"

}

return x

}

function validCase(x) {

let flag = 0

x = x.trim()

console.log("validingCase:" + x)

x = validKey(x)

x = x.trim()

if (x[0] == ":") {

flag++

x = x.slice(1)

x = x.trim()

if (x[0] == "(") {

flag++

x = x.slice(1)

x = x.trim()

x = validSameFields(x)

x = x.trim()

if (x[0] == ")") {

flag++

x = x.slice(1)

x = x.trim()

}

}

}

if (flag != 3) {

throw "Wrong Case"

}

return x

}

function validCases(x) {

x = x.trim()

console.log("validingCases:" + x)

x = validCase(x)

x = x.trim()

if (x[0] == ";") {

x = x.slice(1)

x = validCase(x)

} else {

console.log("validingCases end")

}

x = x.trim()

return x

}

function validSwitchFields(x) {

let flag = 0

x = x.trim()

console.log("validingSwitch:" + x)

if (x.slice(0, 4) == "case") {

flag++

x = x.slice(4)

x = validID(x)

x = x.trim()

if (x.slice(0, 2) == "of") {

flag++

x = x.slice(2)

x = validCases(x)

}

}

if (flag != 2) {

throw "Wrong Switch"

}

x = x.trim()

return x

}

function validStaticFields(x) {

let flag = false

x = x.trim()

console.log("validingStatic:" + x)

x = validSameFields(x)

if (x[0] == ";") {

x = x.slice(1)

x = x.trim()

if (x.slice(0, 4) != "case") {

flag = true

while (flag) {

flag = false

x = validSameFields(x)

if (x[0] == ";") {

x = x.slice(1)

x = x.trim()

if (x.slice(0, 4) != "case") {

flag = true

}

}

}

}

}

return x

}

function validRecord(x){

let flag = 0

x = x.toLowerCase()

x = x.replace(/[\n\t]/g," ")

x = x.trim()

console.log("validingRecord:" + x)

if(x.slice(0,3)=="var"){

flag++

x = x.slice(3)

x = x.trim()

let xcpy

xcpy = x

x = validID(x)

logOnUi("Record name: "+ xcpy.slice(0, (xcpy.length-x.length)))

x = x.trim()

if(x.slice(0,7)==":record"){

flag++

x = x.slice(7)

x = x.trim()

x = validStaticFields(x)

x = x.trim()

if(x.slice(0,4)=="end;"){

flag++

} else {

x = validSwitchFields(x)

if(x.slice(0,4)=="end;"){

flag++

}

}

}

}

if(flag!=3){

throw "Wrong Record"

}

return x

}

// =========================== UI Business-logic ===============================

function validUiCall(){

codeInput = document.getElementById("codeInput")

resultLine = document.getElementById("resultInput")

code = codeInput.value

try{

validRecord(code)

resultLine.value = "Code is correct"

resultLine.className = "bg-success form-control"

}catch(e){

resultLine.value = e

resultLine.className = "bg-danger form-control"

}

}

function logOnUi(x)

{

logArea = document.getElementById("logTextArea")

logArea.value += x + "\n"

}

function resetUi(){

resultLine = document.getElementById("resultInput")

resultLine.value = ""

resultLine.className = "form-control"

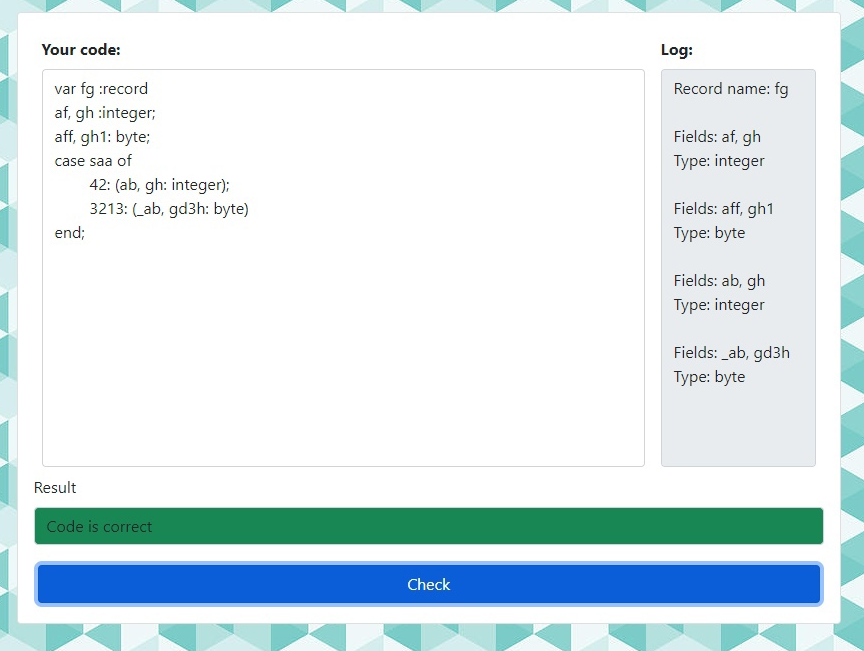
logArea = document.getElementById("logTextArea")

logArea.value = ""

}

Графический интерфейс программы

Вид графического интерфейса представлен на рисунке 9 (использован веб-браузер Chrome).

Рисунок 9 - графический интерфейс программы

Тестирование программы

Тесты, с помощью которых была проверена корректность исполнения программы представлены в таблице 1.

Таблица 1 — тестирование программы

|  |  |  |
| --- | --- | --- |
| *Входные данные* | *Ожидаемый вывод* | *Вывод* |
| var fg :record  af, gh :integer;  aff, gh1: byte;  case saa of  42: (ab, gh: integer);  3213: (\_ab, gd3h: byte)  end; | Record name: fg  Fields: af, gh  Type: integer  Fields: aff, gh1  Type: byte  Fields: ab, gh  Type: integer  Fields: \_ab, gd3h  Type: byte  **Code is correct** | Record name: fg  Fields: af, gh  Type: integer  Fields: aff, gh1  Type: byte  Fields: ab, gh  Type: integer  Fields: \_ab, gd3h  Type: byte  **Code is correct** |
| var fg :record  af, gh :integer;  aff, gh1: byte  end; | Record name: fg  Fields: af, gh  Type: integer  Fields: aff, gh1  Type: byte  **Code is correct** | Record name: fg  Fields: af, gh  Type: integer  Fields: aff, gh1  Type: byte  **Code is correct** |
| var fg :record  af, **2gh** :integer;  aff, gh1: byte  end; | Record name: fg  **Wrong ID** | Record name: fg  **Wrong ID** |
| var fg :record  af, gh :integer;  aff, gh1: byte;  case saa of  **a42**: (ab, gh: integer);  3213: (\_ab, gd3h: byte)  end; | Record name: fg  Fields: af, gh  Type: integer  Fields: aff, gh1  Type: byte  **Wrong Key** | Record name: fg  Fields: af, gh  Type: integer  Fields: aff, gh1  Type: byte  **Wrong Key** |
| var fg :record  af, gh :integer;  aff, gh1: **bte**;  end; | Record name: fg  Fields: af, gh  Type: integer  Fields: aff, gh1  **Wrong Type** | Record name: fg  Fields: af, gh  Type: integer  Fields: aff, gh1  **Wrong Type** |

Контрольные вопросы

1. Дайте определение формального языка и формальной грамматики.

Формальным языком L в алфавите A называют произвольное подмножество множества A\*. Язык можно задать перечислением и правилами продукции.

Грамматика в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита.

Грамматики бывают порождающими и аналитическими. Порождающие грамматики задаются четверкой G = (VT, VN, P, S), где VT — множество терминальных символов, VN — множество нетерминальных символов, P – множество порождающих правил, S – начальный символ.

2. Как определяется тип грамматики по Хомскому?

* тип 0. неограниченные грамматики — возможны любые правила

α → β, где α єV+, βєV\*

* тип 1. контекстно-зависимые грамматики — левая часть может содержать один нетерминал, окруженный «контекстом» (последовательности символов, в том же виде присутствующие в правой части); сам нетерминал заменяется непустой последовательностью символов в правой части.

αXβ → αxβ, где XєVN , xєVТ , α, β є V\*, причем α, β одновременно не являются пустыми, а значит возможность подстановки х вместо символа X определяется присутствием хотя бы одной из подстрок α и β, т. е. контекста;

* тип 2. контекстно-свободные грамматики — левая часть состоит из одного нетерминала, соответственно, подстановки не зависят от контекста.

A → β, где AєVN , βє V\*

* тип 3. регулярные грамматики — эквивалентны конечным автоматам.

A → α, A → αB или A → Bα, где A, ВєVN , α є VT .

3. Поясните физический смысл и обозначения формы Бэкуса–Наура.

Форма Бэкуса-Наура (БНФ) Используется для описания контекстно-свободных формальных грамматик, посредством последовательной замены одних выражений другими. Нетерминальные символы обозначаются в <...>. Используемые операции: «::=» - замена и «|» - «или».

4. Что такое лексический анализ? Какие методы выполнения лексического анализа вы знаете?

При выполнении лексического анализа текст разбивают на «предложения» – операторы языка, а операторы – на «слова», которые применительно к компиляции называют лексемами.

Лексический анализатор выполняет преобразование исходного текста в строку однородных символов. Каждый символ результирующей строки – токен - соответствует слову языка.

Обычно исходный текст разбивается на токены с помощью конечно автомата, в чем и заключается лексический анализ.

В случае метода рекурсивного спуска, лексический и синтаксический анализы не разделяются явным образом.

5. Что такое синтаксический анализ? Какие методы синтаксического анализа вы знаете? К каким грамматикам применяются перечисленные вами методы?

Синтаксический анализ – процесс распознавания конструкций языка в строке токенов.

Метод выполнения синтаксического анализатора определяется типом грамматики языка:

1. для регулярных грамматик используют конечные автоматы;
2. для КС грамматик – автоматы с магазинной памятью (на практике обычно заменяется или рекурсивным спуском или пишется программа с использованием свойств грамматики предшествования).

6. Что является результатом лексического анализа?

Результатом лексического анализа является строка токенов. Каждый токен соответствует слову языка – лексеме и характеризуется набором атрибутов, таких как тип, адрес и т. п.

7. Что является результатом синтаксического анализа?

Кроме распознавания заданной конструкции, результатом лексического анализа является информация об ошибках в выражениях, операторах и описаниях программы.

8. В чем заключается метод рекурсивного спуска?

По синтаксическим диаграммам разрабатываются функции проверки конструкций языка, а затем составляется основная программа начинающая вызов функций с функции, реализующей аксиому языка.

9. Что такое таблица предшествования и для чего она строится?

Таблица предшествования - таблица, показывающая отношения предшествования терминалов.

10. Как с использованием таблицы предшествования осуществляют синтаксический анализ?

Таблица позволяет находить и сворачивать синтаксические основы, если основа не имеет начала или конца, то синтаксический анализатор выдаст ошибку.

**Вывод:** в ходе данной работы были изучены методы лексического и синтаксического анализа контекстно-свободных грамматик.