Министерство образования Республики Молдовы

Технический Университет Молдовы

Факультет Вычислительной техники, информатики и микроэлектроники

Кафедра Автоматики и Информационных Технологий

**Курсовая работа**

По дисциплине: Формальные языки и проектирование компиляторов

Тема: Построение лексического и синтаксического анализатора для языка программирования LISP

Вариант 5

Руководитель: старший преподаватель

Скороходова Татьяна Александрова

Выполнил: ст. группы TI-155 Буянов Евгений

Кишинёв 2016

**Содержание**

Введение………………………………………………….………………………………………………………………..3

1 Цель работы и задание………………………………………..………….........................................4

2 Лексический анализатор……………………………………………………………..……….…...………….6

2.1 Общие теоретические понятия о лексическом анализаторе ………..………………...6

2.2 Регулярная грамматика и конечные автоматы……………………………………………….....8

2.3Выполенение заданий…………………………………………………………………….……………….…..9

3 Восходящий синтаксический анализатор ………………………………………………..….…… 18

3.1 Общие теоретические понятия о восходящем синтаксическом анализаторе…………………………………………………………………………………………………………....18

3.2 Отношения предшествования в грамматиках с простым предшествованием……………………………………………………………………………………….………..18

3.3 Выполнение заданий……………………………………………………. ..….……………………….....20

4 Нисходящий синтаксический анализатор …………………………………………………………25

4.1Общие теоретические понятия о нисходящем синтаксическом анализаторе……………………………………………………………………………………………………………25

4.2 Алгоритмы, необходимые для построения нисходящего синтаксического анализатора……………..…………………………………………………………………………………….........25

4.3 Выполнение заданий………………………………………..…………………………..…………………30

Заключение………………………………………...…………………………………………………..…………….40

Список используемых источников……………………...………….…………………………..………..41

Приложение 1. Листинг программы………………………………………………………...….........42

**Введение**

Компиляция — [трансляция программы](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80), составленной на исходном [языке высокого уровня](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в эквивалентную программу на [низкоуровневом языке](https://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D0%B7%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), близком [машинному коду](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4). Входной информацией для компилятора ([исходный код](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4)) является описание алгоритма или программа на [объектно-ориентированном языке](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

Компиляторы составляют существенную часть программного обеспечения ЭВМ. Это связано с тем, что языки высокого уровня стали основным средством разработки программ. Только малая часть программного обеспечения, требующая особой эффективности, программируется с помощью ассемблеров. В настоящее время распространено довольно много языков программирования. Наряду с традиционными языками, такими, как: С++, С#, Java - широкое распространение получили, так называемые, «универсальные» языки (Паскаль, Си, Модула-2, Ада и другие), а также некоторые специализированные такие как Лисп. Кроме того, большое распространение получили языки, связанные с узкими предметными областями, такие, как входные языки пакетов прикладных программ.

Компилятор работает в несколько фаз, причём в процессе каждой фазы происходит преобразование исходной программы из одного представления в другое. Для выполнения курсовой работы требуется написать программу, которая выполняет лексический и синтаксический анализ входного языка программирования, порождает таблицу лексем с указанием их типов и значений, а также строит синтаксическое дерево. Текст входного языка вводится с клавиатуры или копируется из указанного файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапах лексического и синтаксического анализа. Результатом курсового проекта должна быть программа-анализатор, состоящая из лексического анализатора, разбивающего исходный текст программы на лексемы для последующей их обработки синтаксическим анализатором.

**1 Цель работы и задание**

Целью данной курсовой работы является изучение основных понятий теории регулярных грамматик, ознакомление с назначением и принципами работы лексических анализаторов, получение практических навыков построения сканера на примере заданного простейшего входного языка.

Для выполнения лабораторной работы требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

**Задание:**

На основе определённых конструкций и правил языка LISP построить лексический анализатор, а также восходящий и нисходящий синтаксический анализатор. Основные конструкции языка – атомы и списки. Атом может быть идентификатором и целым числом (символьный и численный атом). Атомы разделяются одним или более пробелов.

Конструкции языка программирования LISP:

<программа> → <S-выражение>

<S-выражение> → <атом>

<S-выражение> → (<список аргументов>

<список аргументов> → <S-выражение>)

<список аргументов> → <S-выражение> <список аргументов>

<атом> → <символьный атом>

<атом> → <численный атом>

**<символьный атом>** - может состоять из буквенных символов ***A, a***, а также цифры (произвольно была взята 9).Вначале символьного атома должен быть обязательно буквенный символ.

**<численный атом>** - может состоять из цифр **0, 1, 2, 3.** С цифры **0** численный атом начинаться не может, однако допустимо чтобы, он состоял только из одного **0**.

Введём следующие обозначения:

**<программа>** - R

**<S-выражение>** - S

**<список аргументов>** - L

**<атом>** - A

**<символьный атом>** - i

**<численный атом>** - n

Тогда, получим грамматику вида:

G = (VN, VT, P, S)

VN = {R, S, L, A}

VT = {i, n, (, ), \_}

P= {

1. R → S
2. S → A
3. S → (\_L
4. L → S\_)
5. L → S\_L
6. A → i
7. A → n

}

**2 Лексический анализатор**

**2.1 Общие теоретические понятия о лексическом анализаторе**

**Лексический анализ** — процесс аналитического разбора входной последовательности символов с целью получения на выходе последовательности символов, называемых «[токенами](http://ru.wikipedia.org/w/index.php?title=%D0%A2%D0%BE%D0%BA%D0%B5%D0%BD_%28%D0%BB%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%29&action=edit&redlink=1)» (подобно группировке букв в словах). Группа символов входной последовательности, идентифицируемая на выходе процесса как токен, называется **лексемой**. В процессе лексического анализа производится распознавание и выделение лексем из входной последовательности символов.

Как правило, лексический анализ производится с точки зрения определённого [формального языка](http://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) или набора языков. Язык, а точнее его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Традиционно принято организовывать процесс лексического анализа, рассматривая входную последовательность символов как поток символов. При такой организации процесс самостоятельно управляет выборкой отдельных символов из входного потока.

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике не может быть идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей *идентификатор токена* (или идентификатор класса токена) и, если нужно, последовательность символов *лексемы*, выделенной из входного потока (строку, число и т. д.).

**Лексический анализатор** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *lexical analyzer* или коротко *lexer*) — это программа или часть программы, выполняющая лексический анализ. Лексический анализатор обычно работает в две стадии: *сканирование* и *оценка*.

На первой стадии, сканировании, лексический анализатор обычно реализуется в виде [конечного автомата](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82), определяемого [регулярными выражениями](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F). В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах. Например, токен «целое число» может содержать любую последовательность десятичных цифр. Во многих случаях первый непробельный символ может использоваться для определения типа следующего токена, после чего входные символы обрабатываются один за другим пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько более сложные и требуют возвратов назад по читаемой последовательности.

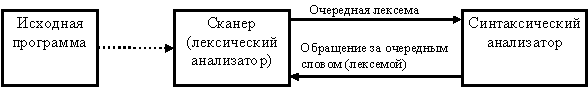
С теоретической точки зрения лексический анализатор не является обязательной, необходимой частью компилятора. Его функции могут выполняться на этапе синтаксического разбора. Однако существует несколько причин, исходя из которых, в состав практически всех компиляторов включают лексический анализ. Эти причины заключаются в следующем:

1. упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокращается объем обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и выкидывает всю незначащую информацию;
2. для выделения в тексте и разбора лексем возможно применять простую, эффективную и теоретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;
3. сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходный программы, структура которого может варьироваться в зависимости от версии входного языка - при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой сканер.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих пробелов, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка.

В простейшем случае фазы лексического и синтаксического анализа могут выполняться компилятором последовательно. Но для многих языков программирования информации на этапе лексического анализа может быть недостаточно для однозначного определения типа и границ очередной лексемы.

В большинстве компиляторов лексический и синтаксический анализаторы - это взаимосвязанные части. Лексический разбор исходного текста в таком варианте выполняется поэтапно так, что синтаксический анализатор, выполнив разбор очередной конструкции языка, обращается к сканеру за следующей лексемой. При этом он может сообщить информацию о том, какую лексему следует ожидать. В процессе разбора может даже происходить “откат назад”, чтобы выполнить анализ текста на другой основе. В дальнейшем будем исходить из предположения, что все лексемы могут быть однозначно выделены сканером на этапе лексического разбора.

Рис.1- Работа синтаксического и лексического анализаторов

**2.2 Регулярная грамматика и конечные автоматы**

Регулярная грамматика – формальная грамматика типа 3 по иерархии Хомского. Регулярные грамматики определяют в точности все регулярные языки, и поэтому эквивалентны конечным автоматам и регулярным выражениям. Регулярные грамматики являются подмножеством контекстно-свободных языков.

Регулярная грамматика может быть задана набором правил, как левая или правая регулярная грамматика. Любая регулярная грамматика может быть преобразована из левой в правую и наоборот.

Любая контекстно-свободная грамматика может быть легко преобразована в вид, в котором правила состоят только из лево-регулярных или право-регулярных (для контекстно-свободных грамматик допустимо наличие тех и других одновременно).

Любая грамматика задается множеством G, которое в свою очередь содержит в себе VN – множество нетерминальных символов, которые задают последовательность вывода VT. Множество VT – множество терминальных символов, из которых состоит получаемое с помощью заданных правил слово. Правила P – множество правил, задающие последовательность вывода терминальных символов для слова. Аксиома – нетерминальный символ, с которого начинается построение слова.

Таким образом, G = (VN, VT, P, S), где S – аксиома.

Конечным автоматом называется формальная система АF = (Q, Σ, δ, q0, F), где Q – конечное непустое множество состояний; Σ – конечный входной алфавит; δ – отображение типа Q × Σ → Q; q0∈Q – начальное состояние; F ⊆ Q – множество конечных состояний.

Конечный автомат может быть детерминированным или недетерминированным. Детерминированный конечный автомат отличается от недетерминированного тем, что у последнего существует более одного перехода по одному символу из множества ∑ в различные состояния из множества Q.

**2.3 Выполнение заданий**

**1 В заданном неформальном описании языка выявить лексемы и для каждого типа лексем построить конечный автомат, допускающие правильные лексемы.**

К лексемам данного языка относятся: *символьные* и *числовые атомы*.

Строим конечные автоматы для данных типов лексем:

*Конечный автомат для символьного атома:*

AF= (Q, ∑, δ, q0, F),

Q = {q0, q1},

∑ = {a, A, 9}, F = {q1}.

δ (q0, a) = q1,

δ (q0, A) = q1,

δ (q1, 9) = q1,

δ (q1, a) = q1,

δ (q1, A) = q1.

Граф:

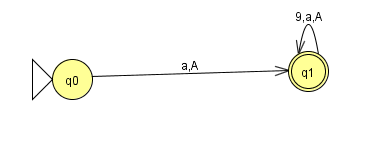


Рис.2 - Граф конечного автомата для символьного атома

Таблица 1- Матрица переходов конечного автомата символьного атома

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Состояния | a | A | 9 | ε |
| [q0] | [q1] | [q1] | -- |  |
| [q1] | [q1] | [q1] | [q1] | Д |

Конечный автомат для числового атома:

AF= (Q, ∑, δ, q0, F),

Q = {q0, q1, q2},

∑ = {0, 1, 2, 3}, F = {q1, q2}.

δ (q0, 0) = q1,

δ (q0, 1) = q2,

δ (q0, 2) = q2,

δ (q0, 3) = q2,

δ (q2, 0) = q2,

δ (q2, 1) = q2,

δ (q2, 2) = q2,

δ (q2, 3) = q2.

Граф:

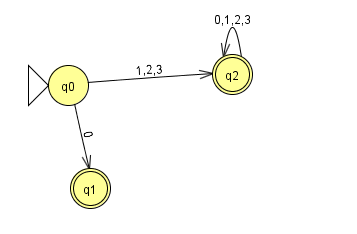


Рис. 3 - Граф конечного автомата для численного атома

Таблица 2 - Матрица переходов конечного автомата числового атома

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Состояния | 0 | 1 | 2 | 3 | Ε |
| [q0] | [q1] | [q2] | [q2] | [q2] |  |
| [q1] |  |  |  |  | Д |
| [q2] | [q2] | [q2] | [q2] | [q2] | Д |

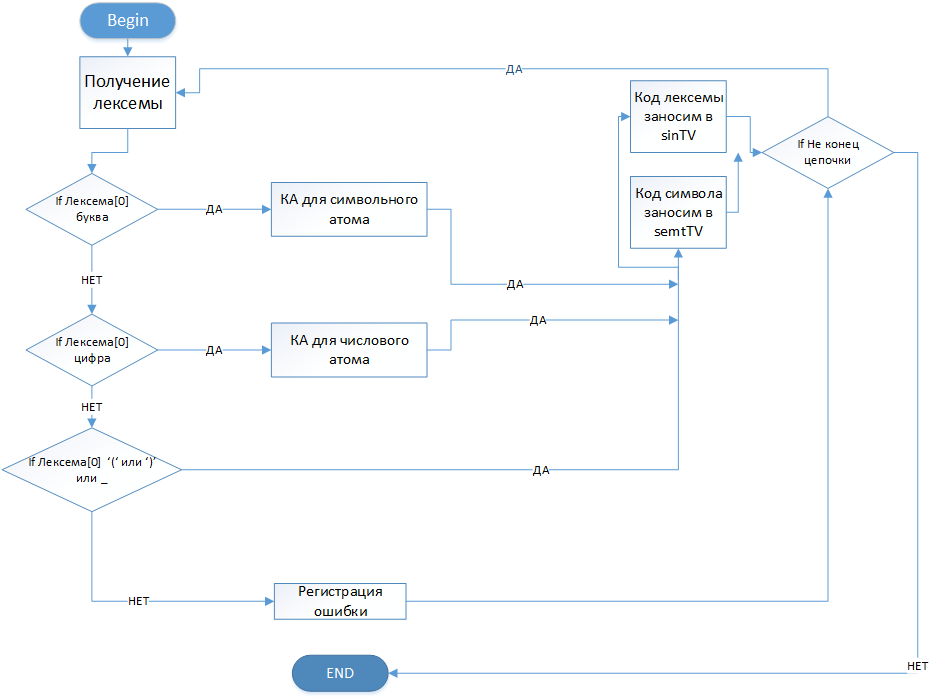
1. **Являются ли построенные конечные автоматы детерминированными?**

Да, каждый из построенных конечный автоматов является детерминированным, так как из каждого состояние есть однозначный переход в другое состояние.

1. **Если какой-либо конечный автомат является недетерминированным, построить эквивалентный ему конечный детерминированный автомат.**

Все полученные конечные автоматы являются детерминированными.

1. **Построить схему лексического анализатора.**



1. **На 5 правильных входных цепочках и трех цепочках, содержащих неправильные лексемы или недопустимые символы, протестировать работу ЛА, построив для этих конкретных цепочек:**

* **Синтаксический вектор,**
* **Семантический вектор,**
* **Заполнение таблиц для каждого вида лексем.**

Все распознанные лексемы (символьные и числовые атомы) для правильных цепочек были занесены в таблицу 3 и таблицу 4.

Таблица 3 - Символьные атомы Таблица 4 - Численные атомы.

|  |  |
| --- | --- |
| Символьный атом | Код |
| a9a9 | 1 |
| A99aA | 2 |
| Aa | 3 |
| a | 4 |
| a99 | 5 |
| AA | 6 |
| a99A | 7 |
| aa | 8 |

|  |  |
| --- | --- |
| Численный атом | Код |
| 1223 | 1 |
| 0 | 2 |
| 12 | 3 |
| 12232 | 4 |
| 1110 | 5 |

В таблице 5 указаны коды лексем, которые использовались для составления семантических векторов.

Таблица 5 - Лексемы

|  |  |
| --- | --- |
| Лексема | Код |
| Символьный атом | 1 |
| Числовой атом | 2 |
| ( | 3 |
| ) | 4 |
| \_ | 5 |

**Правильные цепочки:**

* **1 цепочка**

*a9a9 ( 1223 )*

Проверяем символьный атом на конечном автомате (первый символ буква):

(qo, a9a9) — (q1, 9a9)— (q1, a9)— (q1, 9) — (q1, ε) - Допуск

Проверяем числовой атом на конечном автомате (первый символ цифра):

(qo, 1223) — (q2, 223) — (q2, 23) — (q2, 3) — (q2, ε) - Допуск

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SINTV | 1 | 5 | 3 | 5 | 2 | 5 | 4 |
| SEMTV | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

* **2 цепочка**

*( A99aA ) 0 12*

(qo, *A99aA*) — (q1, *99aA*) — (q1, *9aA*) — (q1, *aA*) — (q1, A) — (q1, ε) - Допуск

(qo, 0)— (q1, ε) - Допуск

(qo, 12) — (q2, 2) — (q2, ε) - Допуск

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SINTV | 3 | 5 | 1 | 5 | 4 | 5 | 2 | 5 | 2 |
| SEMTV | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 3 |

* **3 цепочка**

( 0 12 ) Aa ( a )

(qo, 0) — (q1, ε) - Допуск

(qo, 12)— (q2, 2) — (q2, ε) - Допуск

(qo, Aa) — (q1, a) — (q1, ε) – Допуск

(qo, a) — (q1, ε) - Допуск

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SINTV | 3 | 5 | 2 | 5 | 2 | 5 | 4 | 5 | 1 | 5 | 3 | 5 | 1 | 5 | 4 |
| SEMTV | 0 | 0 | 2 | 0 | 3 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 4 | 0 | 0 |

* **4 цепочка**

0 12232 ( a99 ) a

(qo, 0) — (q1, ε) - Допуск

(qo, *12232*) — (q2, *2232*) — (q2, *232*) |— (q2, *32*) |— (q2, *2*) |— (q2, ε) - Допуск

(qo, a99) |— (q1, 99) |— (q1, 9) |— (q1, ε) - Допуск

(qo, a) |— (q1, ε) - Допуск

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SINTV | 2 | 5 | 2 | 5 | 3 | 5 | 1 | 5 | 4 | 5 | 1 |
| SEMTV | 2 | 0 | 4 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 4 |

* **5 цепочка**

*( AA 0 ( 1110 a99A ) 12 ) aa*

(qo, AA) |— (q1, A) |— (q1, ε) - Допуск

(qo, 0) |— (q1, ε) - Допуск

(qo, *1110*) |— (q2, *110*) |— (q2, *10*) |— (q2, *0*) |— (q2, ε) - Допуск

(qo, *a99A*) |— (q1, *99A*) |— (q1, *9A*) |— (q1, *A*) |— (q1, ε) - Допуск

(qo, 12) |— (q2, 2) |— (q2, ε) - Допуск

(qo, aa) |— (q1, a) |— (q1, ε) - Допуск

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SINTV | 3 | 5 | 1 | 5 | 2 | 5 | 3 | 5 | 2 | 5 | 1 | 5 | 4 | 5 | 2 | 5 | 4 | 5 | 1 |
| SEMTV | 0 | 0 | 6 | 0 | 2 | 0 | 0 | 0 | 5 | 0 | 7 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 8 |

**Неправильные цепочки:**

* **1 цепочка**

*A99a ( 12 ) 32 ( Ab Aa )*

(qo, *A99a*) |— (q1, *99a*) |— (q1, *9a*) |— (q1, *a*) |— (q1, ε) - Допуск

(qo, 12) |— (q2, 2) |— (q2, ε) - Допуск

(qo, 32) |— (q2, 2) |— (q2, ε) - Допуск

(qo, Ab) |— (q1, b) - Тупик!

(qo, *Aa*) |— (q1, *a*) |— (q1, ε) - Допуск

Лексема *Ab* не принадлежит КА для символьных атомов. Вся цепочка содержит одну синтаксическую ошибку.

* **2 цепочка**

*( 30102 aa ) AA 022*

(qo, 30102) |— (q2, 0102) |— (q2, 102) |— (q2, 02) |— (q2, 2) |— (q2, ε) - Допуск

(qo, aa) |— (q1, a) |— (q1, ε) - Допуск

(qo, AA) |— (q1, A) |— (q1, ε) - Допуск

(qo, 022) |— (q1, 22) - Тупик!

Лексема *022* не принадлежит КА для численных атомов. Вся цепочка содержит одну синтаксическую ошибку.

* **3 цепочка**

*aA8 32 ( 99aA aa9 )*

(qo, *aA8*) |— (q1, *A8*) |— (q1, *8*) - Тупик!

(qo, 32) |— (q2, 2) |— (q2, ε) - Допуск

Первый символ цифра, проверяем на КА для численных атомов

(qo, 99aA) - Тупик!

(qo, aa9) |— (q1, a9) |— (q1, 9) |— (q1, ε) - Допуск

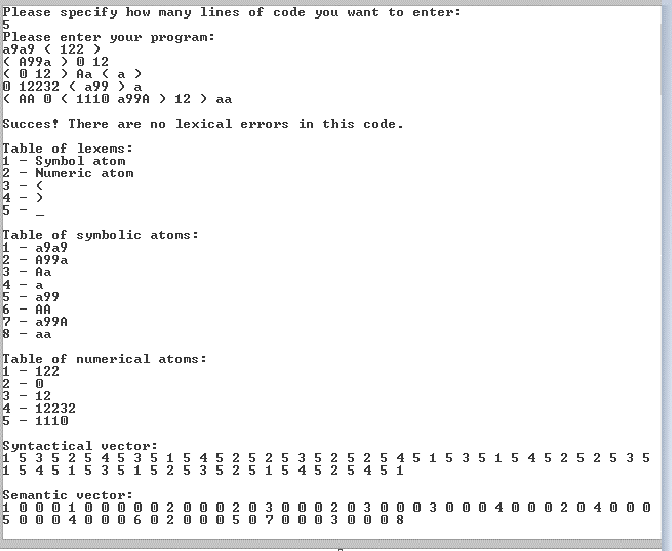
Лексемы *aA8* и *99aA* не принадлежат КА для символьных и числовых атомов. Вся цепочка содержит две синтаксические ошибки.

1. **Разработать программу для лексического анализатора и продемонстрировать ее работу для построенных входных цепочек.**

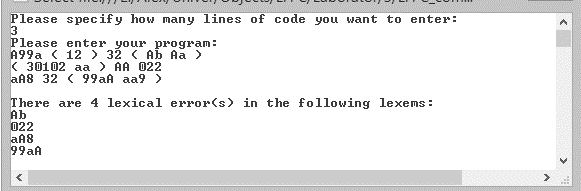
Код программы для лексического анализатора в Приложении 1.

Результаты работы программы:

**Правильные цепочки**



**Неправильные цепочки**

****

**3 Восходящий синтаксический анализатор**

**3.1 Общие теоретические понятия о восходящем синтаксическом анализаторе**

При восходящем грамматическом разборе правила подстановки применяются для свертки (приведения) всего исходного предложения к начальному символу грамматики. С этой целью в приводимой строке последовательно отыскивается основа – самая левая часть строки, которую по одному из правил подстановки следует заменить на нетерминальный, и выполняется эта замена.

Например, рассмотрим грамматику арифметических выражений с использованием идентификаторов и операций + и \*:

S → T | T+S a\***b**+c  **a\*T**+c  T+**c**  T+**T**  **T+S**  S

T → ид | ид\*T

Основная проблема – нахождение основы. Так, если первой выполнить свертку a\*b+c  T\*b+c, т.е. за основу принять символ ‘a’, то в дальнейшем полностью свернуть предложение к начальному символу грамматики уже не удастся. Нахождение основы может быть выполнено различными методами, в частности, путем определения отношений предшествования между символами.

**3.2 Отношения предшествования в грамматиках с простым предшествованием**

Между любыми двумя последовательными символами приводимой строки p и q могут существовать три отношения:

p<q, если q – самый левый символ некоторой основы

p>q, если p – самый правый символ некоторой основы

p=q, если символы p и q находятся внутри основы

Отношения <, >, = называются отношениями предшествования.

Пусть p, q – любые символы (терминальные или нетерминальные), U, C, D – нетерминальные символы, x, y, z, w – любые строки, возможно пустые. Грамматика называется *грамматикой с простым предшествованием*, если:

1. Не содержит e-продукций
2. Не позволяет более одного вывода A \*=> A
3. Не содержит продукции с одинаковыми правыми частями
4. Между 2-мя символами не более, чем одно отношение предшествования.

Матрицу предшествования грамматики можно построить, опираясь непосредственно на определения отношений предшествования, но удобнее воспользоваться двумя дополнительными множествами - множеством крайних левых и множеством крайних правых символов относительно нетерминалов грамматики.

Для определения отношений предшествования предварительно для каждого нетерминального символа U находятся два множества:

PRIM(U) – множество самых левых символов в разложениях U, т.е. множество таких символов q, что существует вывод Uqz;

ULTIM(U) – множество самых правых символов в разложениях U, т.е. множество таких символов p, что существует вывод Uzp.

Построение множеств PRIM(U), ULTIM(U) состоит из следующих шагов:

1. для каждого правила вида U → z в PRIM(U) заносится самый левый символ правой части, в ULTIM(U) – самый правый символ;
2. если PRIM(U) содержит нетерминальные символы U1, U2, ..., то PRIM(U) дополняется символами, входящими в L(U1), L(U2) ... и не входящими в PRIM(U). Аналогично дополняется ULTIM(U);
3. шаг 2 повторяется, пока PRIM(U), ULTIM(U) не перестанут изменяться.
4. Для вывода отношений между символами просматриваются правые части правил:

* для каждой последовательной пары символов pq принимается p=q;
* для каждого сочетания pU принимается p<q, qPRIM(U);
* для каждого сочетания Uq принимается p>q, pULTIM(U);
* для каждого сочетания CD принимается p>q, pULTIM(U) и qPRIM(U).

**3.3Выполнение заданий**

1. **Построить отношения простого предшествования для контекстно-свободной грамматики.**

Построим отношения простого предшествия

Таблица 6 - Множества PRIM(U) и ULTIM(U)

|  |  |  |
| --- | --- | --- |
| U | PRIM(U) | ULTIM(U) |
| R | S A, (, i, n A,(, i, n | S A, L, i n ) S,A,L,i n, ) |
| S | A,( (, i, n A, (, i, n | A, L i, n, ) A, L,i, n, ) |
| L | S A, (, i, n S, A,(, i, n | ) ) |
| A | i, n i, n | i, n i, n |

Таблица 7 - Таблица простого предшествования

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | i | n | ( | ) | \_ | R | S | L | A | $ |
| i |  |  |  |  | > |  |  |  |  | > |
| n |  |  |  |  | > |  |  |  |  | > |
| ( |  |  |  |  | = |  |  |  |  |  |
| ) |  |  |  |  | > |  |  |  |  | > |
| \_ | < | < | < | = |  |  | < | = | < |  |
| R |  |  |  |  |  |  |  |  |  |  |
| S |  |  |  |  | = |  |  |  |  | > |
| L |  |  |  |  | > |  |  |  |  | > |
| A |  |  |  |  | > |  |  |  |  | > |
| $ | < | < | < |  |  |  | < |  | < |  |

Наша грамматика является грамматикой простого предшествования, т.к. между любыми двумя символами только одно отношение предшествования или оно отсутствует вообще.

1. **Проверить работу синтаксического анализатора на случайных цепочках.**

**Проверяем правильную цепочку**

w = (\_n\_i\_)

[ $ , (\_n\_i\_)$ , Ɛ ] [ $<( , \_n\_i\_)$ , Ɛ ] [ $<(=\_ , n\_i\_)$ , Ɛ ] [ $<(=\_<n , \_i\_)$ , Ɛ ]

7. A->n

[ $<(=\_<A , \_i\_)$ , 7 ] [ $<(=\_<S , \_i\_)$ , 72 ] [ $<(=\_<S=\_ , i\_)$ , 72 ]

2. S->A

[ $<(=\_<S=\_<i , \_)$ , 72 ] [ $<(=\_<S=\_<A , \_)$ , 726 ]

2. S->A

6. A->i

[ $<(=\_<S=\_<S , \_)$ , 7262 ] [ $<(=\_<S=\_<S=­\_ , )$ , 7262 ]

[ $<(=\_<S=\_<S=­\_=) , $ , 7262 ] [ $<(=\_<S=\_=L , $ , 72624 ]

5. L->S\_L

4. L->S\_)

[ $<(=\_<S=\_=L , $ , 726245 ] [ $<S , $ , 7262453 ] [ $ R , $ , 72624531 ] - Допуск!

1. R->S

3. S->(\_L

***Неправильная цепочка:***

w = (\_ \_n\_)

[ $ , (\_ \_n\_)$ , e ] [ $<( , \_ \_n\_)$ , e ] [ $<(=\_ , \_n\_)$ , e ] |— ОШИБКА!

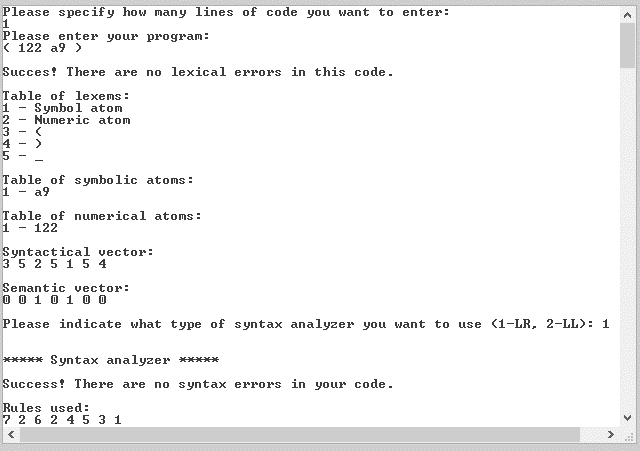
В таблице предшествования отсутствует отношение для символов \_ и \_.

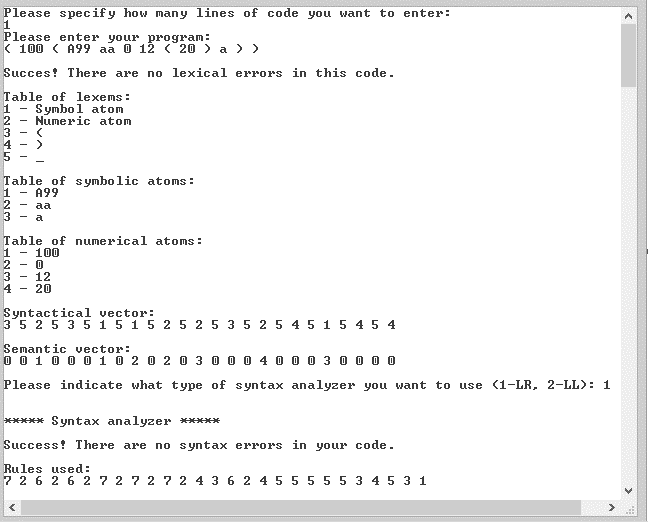
Таким образом, в цепочке была найдена синтаксическая ошибка.

1. **Разработать программу для восходящего синтаксического анализатора и продемонстрировать ее работу для построенных входных цепочек**

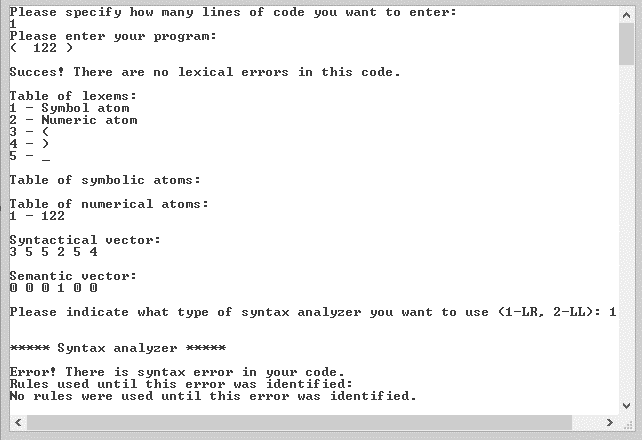
Код программы для лексического анализатора в Приложении 1.

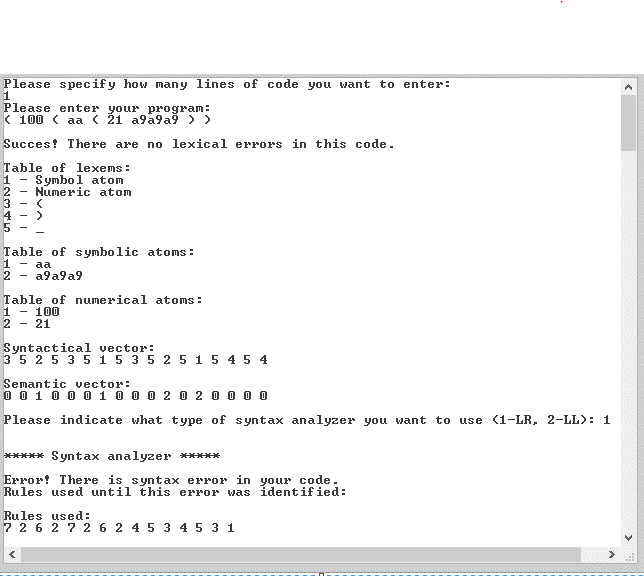
Демонстрация работы программы:

**Правильные цепочки**



***Неправильные цепочки:***



****

**4 Нисходящий синтаксический анализатор**

**4. 1 Общие теоретические понятия о нисходящем синтаксическом анализаторе**

**Синтаксический LL-анализатор** (**LL parser**) — нисходящий [синтаксический анализатор](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80) для подмножества [контекстно-свободных грамматик](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82%D0%BD%D0%BE-%D1%81%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0). Он анализирует входной поток слева направо, и строит [левый вывод](http://ru.wikipedia.org/w/index.php?title=%D0%9B%D0%B5%D0%B2%D1%8B%D0%B9_%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4&action=edit&redlink=1) грамматики. Класс грамматик, для которых можно построить LL-анализатор, известен как [LL-грамматики](http://ru.wikipedia.org/wiki/LL-%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0).

LL(1)-грамматики очень распространены, потому что соответствующие им LL-анализаторы просматривают поток только на один символ вперед при принятии решения о том, какое правило грамматики необходимо применить.

**4.2 Алгоритмы, необходимые для построения нисходящего синтаксического анализатора**

При построении таблицы предсказывающего анализатора (MAP) нам потребуются две функции - FIRST и FOLLOW.

Пусть G = (VN, VN, P, S) - КС-грамматика. Для http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gif- произвольной цепочки, состоящей из символов грамматики, определим FIRST(http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gif) как множество терминалов, с которых начинаются строки, выводимые из http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gif. Если http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gifhttp://citforum.ru/programming/theory/serebryakov/cmsy10-29.gif\*e, то e также принадлежит FIRST(http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gif).

Определим FOLLOW(A) для нетерминала A как множество терминалов a, которые могут появиться непосредственно справа от A в некоторой сентенциальной форме грамматики, т.е. множество терминалов a таких, что существует вывод вида S http://citforum.ru/programming/theory/serebryakov/cmsy10-29.gif\*http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gifAahttp://citforum.ru/programming/theory/serebryakov/cmmi10-c.gif для некоторых http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gif, http://citforum.ru/programming/theory/serebryakov/cmmi10-c.gifhttp://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif(N http://citforum.ru/programming/theory/serebryakov/cmsy10-5b.gifT)\*. Заметим, что между A и a в процессе вывода могут находиться нетерминальные символы, из которых выводится e. Если A может быть самым правым символом некоторой сентенциальной формы, то $ также принадлежит FOLLOW(A).

Рассмотрим алгоритмы построение множества FIRST.

**Алгоритм вычисления FIRST(X) для символов грамматики.**

Вход. КС-грамматика G = (VN, VN, P, S).

Выход. Множество FIRST(X) для каждого символа X http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif( Vn http://citforum.ru/programming/theory/serebryakov/cmsy10-5b.gifVT).

Шаги:

* Если X - терминал, то положить FIRST(X) = {X}; если X - нетерминал, положить FIRST(X) = http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gif.
* Если в P имеется правило X http://citforum.ru/programming/theory/serebryakov/cmsy10-21.gife, то добавить e к FIRST(X).
* Пока ни к какому множеству FIRST(X) нельзя уже будет добавить новые элементы, выполнять:

если X - нетерминал и имеется правило вывода X http://citforum.ru/programming/theory/serebryakov/cmsy10-21.gifY 1Y 2...Y k, то включить a в FIRST(X), если a http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gifFIRST(Y i) для некоторого i, 1 http://citforum.ru/programming/theory/serebryakov/msam10-36.gifi http://citforum.ru/programming/theory/serebryakov/msam10-36.gifk, и e принадлежит всем FIRST(Y 1), ..., FIRST(Y i-1), то есть Y 1...Y i-1 http://citforum.ru/programming/theory/serebryakov/cmsy10-29.gif\*e. Если e принадлежит FIRST(Y j) для всех j = 1, 2, ..., k, то добавить e к FIRST(X).

**Алгоритм вычисления FIRST(α) для цепочки.**

Вход. КС-грамматика G = (VN, VT, P, S).

Выход. Множество FIRST(X1X2...Xn), Xi http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif(Vn http://citforum.ru/programming/theory/serebryakov/cmsy10-5b.gifVT).

Шаги:

* При помощи предыдущего алгоритма вычислить FIRST(X) для каждого X http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif(Vn http://citforum.ru/programming/theory/serebryakov/cmsy10-5b.gifVT).
* Положить FIRST(X1X2...Xn) = http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gif.
* Добавить к FIRST(X1X2...Xn) все не e-элементы из FIRST(X1). Добавить к нему также все не e-элементы из FIRST(X2), если e http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gifFIRST(X1), не e-элементы из FIRST(X3), если e принадлежит как FIRST(X1), так и FIRST(X2), и т.д. Наконец, добавить цепочку e к FIRST(X1X2...Xn), если e http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gifFIRST(Xi) для всех i.

Рассмотрим алгоритм вычисления функции FOLLOW.

**Алгоритм вычисления FOLLOW(A) для нетерминалов грамматики**.

Вход. КС-грамматика G = (VN, VN, P, S).

Выход. Множество FOLLOW(X) для каждого символа X http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif Vn.

Шаги:

* Положить FOLLOW(X) = http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gifдля каждого символа X http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gifVn.
* Добавить $ к FOLLOW(S).
* Если в P eсть правило вывода A http://citforum.ru/programming/theory/serebryakov/cmsy10-21.gifhttp://citforum.ru/programming/theory/serebryakov/cmmi10-b.gifBhttp://citforum.ru/programming/theory/serebryakov/cmmi10-c.gif, где http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gif, http://citforum.ru/programming/theory/serebryakov/cmmi10-c.gifhttp://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif(Vn http://citforum.ru/programming/theory/serebryakov/cmsy10-5b.gifVT)\* то все элементы из FIRST(http://citforum.ru/programming/theory/serebryakov/cmmi10-c.gif), за исключением e, добавить к FOLLOW(B).
* Пока ничего нельзя будет добавить ни к какому множеству FOLLOW(X), выполнять:

если в P есть правило A http://citforum.ru/programming/theory/serebryakov/cmsy10-21.gifhttp://citforum.ru/programming/theory/serebryakov/cmmi10-b.gifB или A http://citforum.ru/programming/theory/serebryakov/cmsy10-21.gifhttp://citforum.ru/programming/theory/serebryakov/cmmi10-b.gifBhttp://citforum.ru/programming/theory/serebryakov/cmmi10-c.gif, http://citforum.ru/programming/theory/serebryakov/cmmi10-b.gif, http://citforum.ru/programming/theory/serebryakov/cmmi10-c.gifhttp://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif(Vn http://citforum.ru/programming/theory/serebryakov/cmsy10-5b.gifVT)\*, где FIRST(http://citforum.ru/programming/theory/serebryakov/cmmi10-c.gif) содержит e (т.е. http://citforum.ru/programming/theory/serebryakov/cmmi10-c.gifhttp://citforum.ru/programming/theory/serebryakov/cmsy10-29.gif\*e), то все элементы из FOLLOW(A) добавить к FOLLOW(B).

**Множество управляющих символов.**

Пусть мы имеем контекстно-свободную грамматику G = (VN, VN, P, S) и A -> αi € P, тогда множество управляющих символов:

SD(A->α) =

|  |
| --- |
| (FIRST(α) \ {e}) http://citforum.ru/programming/theory/serebryakov/cmsy10-5b.gif FOLLOW(A), если α \*=> e |
| FIRST(α), если α \*≠> e |

***Теорема***: Контекстно-свободная грамматика G является LL(1) грамматикой, тогда и только тогда, когда для любого A http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif Vn и всех правил вида

A -> α1 A -> α2 A -> α3 … A -> αn

Выполняется условие:

SD(A -> αi) ^ SD(A -> αj) = http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gif

**Алгоритм построения таблицы предсказывающего анализатора (MAP).**

Вход. КС-грамматика G = (VN, VT, P, S).

Выход. Таблица MAP[A, a] предсказывающего анализатора, A http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gifVn, a http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gifVn http://citforum.ru/programming/theory/serebryakov/cmsy10-5b.gif{$}.

Шаги:

1. T(a, a) = V
2. T($, $) = D
3. Для всех b http://citforum.ru/programming/theory/serebryakov/cmsy10-32.gif SD(i, A -> α) то определяем T(A, b) = i
4. Все остальные клетки таблицы ERROR.
   1. **Выполнение заданий**

**1 Проверить является ли данная контекстно-свободная грамматика LL(1) грамматикой.**

Строим множество FIRST(X)

Таблица 8 - Множество FIRST(X)

|  |  |
| --- | --- |
| X | FIRST(X) |
| R | S A ( i n ( i n |
| S | A ( i n ( i n |
| L | S A ( i n ( i n |
| A | i n i n |

Определим множество управляемых символов:

Таблица 9 - Множества FIRST(α) и SD(A->α)

|  |  |  |  |
| --- | --- | --- | --- |
| № | A -> α | FIRST(α) | SD(A->α) |
| 1 | R->S | S A ( i n ( i n | ( i n |
| 2 | S->A | A i n i n | i n |
| 3 | S->(\_L | ( ( | ( |
| 4 | L->S\_) | S A ( i n ( i n | ( i n |
| 5 | L->S\_L | S A ( i n ( i n | ( i n |
| 6 | A->i | i i | i |
| 7 | A->n | n n | n |

SD(S->A) ∩ SD(S -> (\_L) = {i, n} ∩ { ( } = http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gif

SD(L -> S\_)) ∩ SD(L -> S\_L) = { (, i, n} ∩ { (, i, n } ≠ http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gif => Данная грамматика не является LL(1) грамматикой

Данный конфликт возник из-за того, что мы имеем правила с общим префиксом:

|  |
| --- |
| 4. L->S\_) |
| 5. L->S\_L |

1. **Если данная грамматика не является LL(1) грамматикой, преобразовать её в LL(1)**

Воспользуемся алгоритмом исключения общего префикса из продукций:

Правила типа A->αβ1 A->αβ2  …. A->αβn

Заменяем их на правила вида:

A->αX, где X – новый Vn тогда:

X->β1 X->β2  …. X->βn

Получаем новую грамматику:

G = (VN', VT, P’, R).

P’ = {

1. R → S
2. S → A
3. S → (\_L
4. L → SX
5. X → \_)
6. X → \_L
7. A → i
8. A → n }

Замечаем, что мы снова получили общий префикс у двух правил

1. X → \_)
2. X → \_L

Вводим ещё один Vn символ – Y и следуем алгоритму:

Получаем новую грамматику:

G = (VN'', VT, P’’, R).

VN’’ = { R, S, L, X, Y, A }

P’’ = {

1. R → S
2. S → A
3. S → (\_L
4. L → SX
5. X → \_Y
6. Y → )
7. Y → L
8. A → i
9. A → n }

Проверим изменённую грамматику:

Строим множество FIRST(X)

Таблица 10 - Множество FIRST(X)

|  |  |
| --- | --- |
| X | FIRST(X) |
| R | S A ( i n ( i n |
| S | A ( i n ( i n |
| L | S A ( i n ( i n |
| X | \_ \_ |
| Y | ) L S A ( i n ( i n |
| A | i n i n |

Определим множество управляемых символов:

Таблица 11 - Множества FIRST(α) и SD(A->α)

|  |  |  |  |
| --- | --- | --- | --- |
| № | A -> α | FIRST(α) | SD(A->α) |
| 1 | R->S | S A ( i n ( i n | ( i n |
| 2 | S->A | A i n i n | i n |
| 3 | S->(\_L | ( ( | ( |
| 4 | L->SX | S A ( i n ( i n | ( i n |
| 5 | X->\_Y | \_ \_ | \_ |
| 6 | Y->) | ) ) | ) |
| 7 | Y->L | L S A ( i n ( i n | ( i n |
| 8 | A->i | i i | I |
| 9 | A->n | n n | N |

SD(S->A) ∩ SD(S -> (\_L) = {i, n} ∩ { ( } = http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gif

SD(Y ->) ) ∩ SD(Y -> L) = { ) } ∩ { (, i, n } = http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gif

SD(A -> i ) ∩ SD(A -> n) = { i } ∩ { n } = http://citforum.ru/programming/theory/serebryakov/msbm10-3f.gif

* Изменённая грамматика, является LL(1) грамматикой!

1. **Построить MAP таблицу**

Таблица 12 - MAP таблица

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | n | ( | ) | \_ | $ |
| i | V |  |  |  |  |  |
| n |  | V |  |  |  |  |
| ( |  |  | V |  |  |  |
| ) |  |  |  | V |  |  |
| \_ |  |  |  |  | V |  |
| R | 1 | 1 | 1 |  |  |  |
| S | 2 | 2 | 3 |  |  |  |
| L | 4 | 4 | 4 |  |  |  |
| X |  |  |  |  | 5 |  |
| Y | 7 | 7 | 7 | 6 |  |  |
| A | 8 | 9 |  |  |  |  |
| $ |  |  |  |  |  | Д |

**4 Проверить правильность построения таблицы MAP на примере случайных цепочек.**

При проверке цепочек, используем конфигурации:

Конфигурация: (стек, входная последовательность символов, вывод правил)

1. Начальная конфигурация: (S $, w$, e)
2. При ($, $, П) – Допуск
3. (aj, aw, П) (j1, w1, П)
4. (Aγ1 , aw1, П) (αγ1, aw1, П)

Все остальные комбинации ошибочны

**Демонстрация на правильной цепочке:**

w = (\_n\_i\_)

[ R$ , (\_n\_i\_)$ , e ] [ S$ , (\_n\_i\_)$ , 1 ] [ (\_L$ , (\_n\_i\_)$ , 13 ] [ \_L$ , \_n\_i\_)$ , 13 ] [ L$ , n\_i\_)$ , 13 ] [ SX$ , n\_i\_)$ , 134 ] [ AX$ , n\_i\_)$ , 1342 ] [ nX$ , n\_i\_)$ , 13429 ] [ X$ , \_i\_)$ , 13429 ] [ \_Y$ , \_i\_)$ , 134295 ] [ Y$ , i\_)$ , 134295 ] [ L$ , i\_)$ , 1342957 ] [ SX$ , i\_)$ , 13429574 ] [ AX$ , i\_)$ , 134295742 ] [ iX$ , i\_)$ , 1342957428 ] [ X$ , \_)$ , 1342957428 ] [ \_Y$ , \_)$ , 13429574285 ] [ Y$ , )$ , 13429574285 ] [ )$ , )$ , 134295742856 ] [ $ , $ , 134295742856 ] - Допуск!

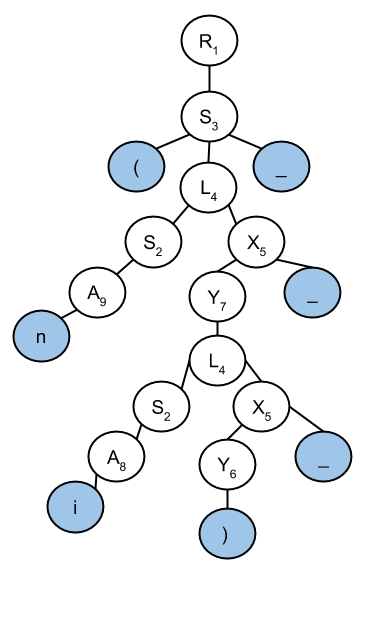


Рис. 4 - Дерево вывода цепочки на нисходящем синтаксическом анализаторе для правильной цепочки

**Демонстрация на неправильной цепочке:**

w = (\_ \_n\_)

[ R$ , (\_ \_n\_)$ , e ] [ S$ , (\_ \_n\_)$ , 1 ] [ (\_L$ , (\_ \_n\_)$ , 13 ] [ \_L$ , \_ \_n\_i\_)$ , 13 ] [ L$ , \_n\_i\_)$ , 13 ]

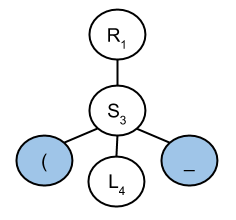


Рис. 5 - Дерево вывода цепочки на нисходящем синтаксическом анализаторе для неправильной цепочки

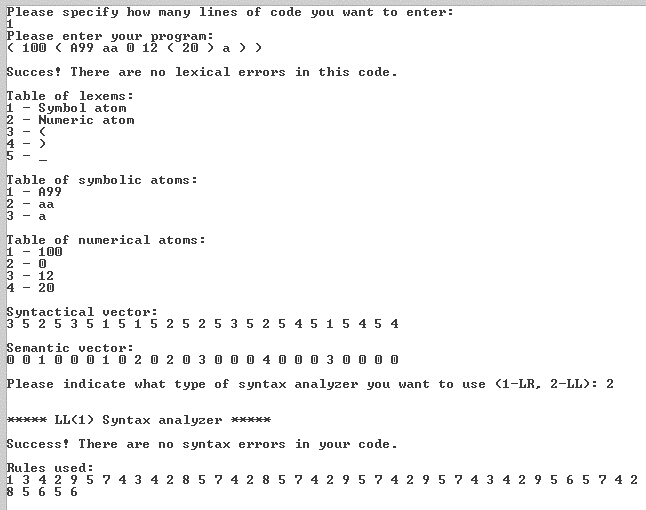
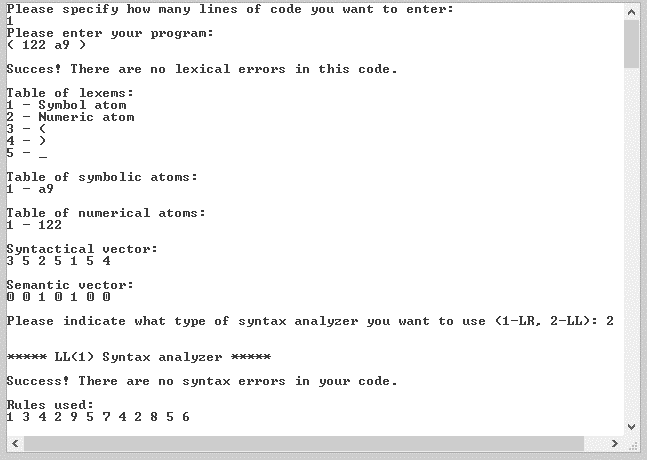
Ошибка! В таблице MAP отсутствует значение для символов **L** и **\_**

Цепочка содержит синтаксическую ошибку.

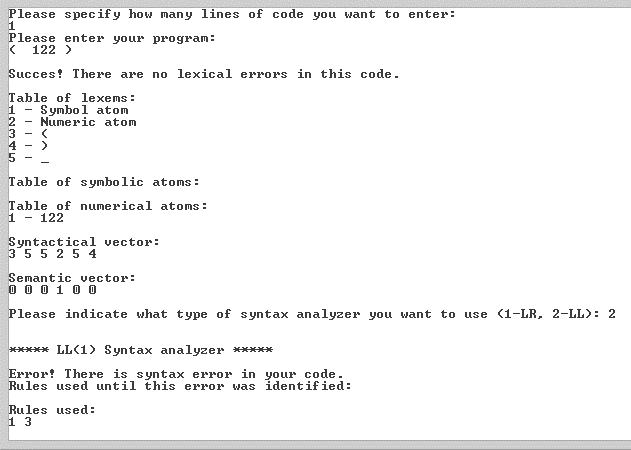
1. **Разработать программу для нисходящего синтаксического анализатора и продемонстрировать ее работу для построенных входных цепочек.**

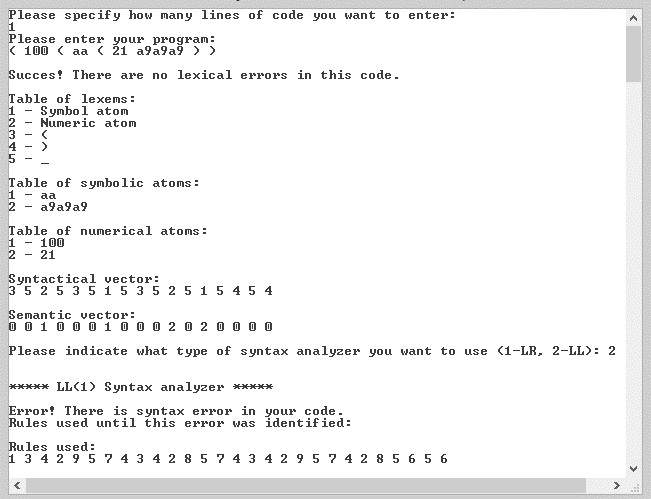
Код программы для лексического анализатора в Приложении 1.

**Демонстрация для правильных цепочек**

****

**Демонстрация для неправильных цепочек:**





**Заключение**

Выполнив курсовую работу, я познакомился и изучил процессы построения лексического и синтаксического анализатора для входных цепочек языка LISP.

Для лексического анализатора были построены конечные автоматы для распознавания каждого типа лексем. Выходными данными лексического анализатора является таблица лексем, служащая входными данными для синтаксического анализатора. Подобная перекодировка исходной программы, которая была реализована в ходе данной курсовой работы, позволяет другим блокам транслятора эффективно работать, оперируя символами фиксированной длины. К примеру, синтаксический анализатор будет работать только с синтаксическим вектором, который представляет собой последовательность целых чисел (кодов лексем).

Синтаксический анализ позволяет проверять соответствие структуры исходного текста заданной грамматике входного языка. К тому же, синтаксический анализ позволяет обнаружить синтаксические ошибки во входной программе. При наличии ошибки пользователю выдается сообщение с указанием местоположения ошибки в исходном тексте, более того, производится анализ типа обнаруженной ошибки.

В восходящем синтаксическом анализаторе была использована построенная таблица предшествования. В свою очередь для нисходящего анализатора была построена таблица MAP. По данным таблицам производился анализ входных цепочек с помощью конфигураций.

Также было замечено, что для нисходящего синтаксического анализатора легче представить его программную модель и проверять входные цепочки.

Отдельные части компилятора, разработанные данной курсовой работе, дают представление о технике и методах, которые лежат в основе построения компиляторов.

**Список используемых источников**

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1: Синтаксический анализ. М.: Мир, 1978. – 612 с.
2. Касьянов В.Н., Поттосин И.В. Методы построения трансляторов. - Новосибирск, Наука, 1986.
3. Хантер Р. Проектирование и конструирование компиляторов. Москва, Финансы и статистика, 1984.
4. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. Москва, Мир, 1979
5. Пратт Т. Языки программирования. Разработка и реализация. Москва, Мир, 1979.
6. Братчиков И. Л. Синтаксис языков программирования. М.: Мир, 1975. – 232 с.
7. Гинзбург С. Математическая теория контекстно-свободных языков. М.: Мир, 1970. – 326 с.
8. Лаллеман Ж. Полугруппы и комбинаторные приложения. М.: Мир, 1985. – 440 с.
9. Хопкрофт Дж. Э., Мотвани Р., Ульман Дж. Д. Введение в теорию автоматов, языков и вычислений, 2-е изд. М.: Вильямс, 2002. – 528 с.
10. [Основы компиляции](http://www.codenet.ru/progr/compil/cmp/intro.php) (рус.) http://www.codenet.ru/progr/compil/cmp/intro.php
11. [В.С. Фомичев. Формальные языки, грамматики и автоматы](http://www.eltech.ru/misc/edu/Index.htm) (рус.) http://www.eltech.ru/misc/edu/Index.htm
12. [Основы разработки трансляторов](http://www.softcraft.ru/translat/lect/content.shtml) (рус.) http://www.softcraft.ru/translat/lect/content.shtml
13. [Компиляторы](http://directory.google.com/Top/World/Russian/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D1%8B/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80%D1%8B/) (рус.) http://directory.google.com/Top/World/Russian/Компьютеры/Программирование/Компиляторы
14. <http://www.mathrix.hut.ru/>
15. Ссылки на книгу Карпова по формальным языкам (формат pdf).  
    Всего глав 7.  
    <http://dcn.nord.nw.ru/bank/compilers/Glava1.pdf>

**Приложение 1:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LFPC\_compilator

{

class LexicalAnalyzer

{

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace LFPC\_compilator

{

class Automaton

{

private int state;

private char symbol;

public Automaton()

{

state = 0;

}

public bool ScanSymbolAtom(string word)

{

state = 0;

for (int i = 0; i < word.Length; i++)

{

symbol = word[i];

switch (state)

{

case 0:

switch (symbol)

{

case 'a':

case 'A':

state = 1;

break;

default:

return false;

}

break;

case 1:

switch (symbol)

{

case 'a':

case 'A':

case '9':

state = 2;

break;

default:

return false;

}

break;

case 2:

switch (symbol)

{

case 'a':

case 'A':

case '9':

break; // State remains unchanged.

default:

return false;

}

break;

default:

return false; // Should never happen because our program controls the states.

}

}

if ((state == 1) || (state == 2))

{

return true;

}

else

{

return false;

}

}

public bool ScanNumericAtom(string number)

{

state = 0;

for (int i = 0; i < number.Length; i++)

{

symbol = number[i];

switch (state)

{

case 0:

switch (symbol)

{

case '0':

state = 1;

break;

case '1':

case '2':

case '3':

state = 2;

break;

default:

return false;

}

break;

case 1:

return false; // 0 can be only as single number, and can't be used to start a number.

case 2:

switch (symbol)

{

case '0':

case '1':

case '2':

case '3':

state = 3;

break;

default:

return false;

}

break;

case 3:

switch (symbol)

{

case '0':

case '1':

case '2':

case '3':

break; // State remains unchanged.

default:

return false;

}

break;

default:

return false;

}

}

if ((state == 1) || (state == 2) || (state == 3))

{

return true;

}

else

{

return false;

}

}

}

}

private Automaton automaton = new Automaton();

// Vectors

private List<int> sintVector = new List<int>();

private List<int> semtVector = new List<int>();

// Tables.

private Dictionary<string, int> symbolicAtomsTable = new Dictionary<string, int>();

private int symbAtomsTableCount = 1;

private Dictionary<string, int> numericAtomsTable = new Dictionary<string, int>();

private int numAtomsTableCount = 1;

public LexicalAnalyzer()

{ }

public void Analyze()

{

int errorsCount = 0;

// Keep strings containing errors to show them later.

List<string> errorLexems = new List<string>();

Console.WriteLine("Please specify how many lines of code you want to enter:");

int numOfLines = int.Parse(Console.ReadLine());

Console.WriteLine("Please enter your program:");

StringBuilder codeLines = new StringBuilder();

for (int i = 0; i < numOfLines; i++)

{

codeLines.Append(Console.ReadLine() + ' ');

}

codeLines.Remove(codeLines.Length - 1, 1); // Remove last space.

string code = codeLines.ToString();

string[] lexems = code.Split(' ');

foreach (string lexeme in lexems)

{

if (Char.IsLetter(lexeme[0]))

{

// Lexeme was accepted.

if (automaton.ScanSymbolAtom(lexeme))

{

sintVector.Add(1);

// If this symbolic atom is new, add it to the dictionary.

if (!symbolicAtomsTable.ContainsKey(lexeme))

{

symbolicAtomsTable.Add(lexeme, symbAtomsTableCount++);

}

semtVector.Add((symbolicAtomsTable.Single(a => a.Key == lexeme)).Value);

}

else

{

errorsCount++;

errorLexems.Add(lexeme);

}

}

else if (Char.IsDigit(lexeme[0]))

{

// Lexeme was accepted.

if (automaton.ScanNumericAtom(lexeme))

{

sintVector.Add(2);

// If this numeric atom is new, add it to the dictionary.

if (!numericAtomsTable.ContainsKey(lexeme))

{

numericAtomsTable.Add(lexeme, numAtomsTableCount++);

}

semtVector.Add((numericAtomsTable.Single(a => a.Key == lexeme)).Value);

}

else

{

errorsCount++;

errorLexems.Add(lexeme);

}

}

else if (lexeme[0] == '(')

{

sintVector.Add(3);

semtVector.Add(0);

}

else if (lexeme[0] == ')')

{

sintVector.Add(4);

semtVector.Add(0);

}

else if (lexeme[0] == ' ')

{

sintVector.Add(5);

semtVector.Add(0);

}

else

{

errorsCount++;

errorLexems.Add(lexeme);

}

}

if (errorsCount != 0)

{

Console.WriteLine("\nThere are {0} syntax error(s) in the following lexems: ", errorsCount);

foreach (string lex in errorLexems)

{

Console.WriteLine(lex);

}

return;

}

Console.WriteLine("\nSucces! There are no syntax errors in this code.");

Console.WriteLine("\nTable of lexems: ");

Console.WriteLine("1 - Symbol atom\n2 - Numeric atom\n3 - (\n4 - )");

Console.WriteLine("\nTable of symbolic atoms:");

foreach (KeyValuePair<string, int> pair in symbolicAtomsTable)

{

Console.WriteLine("{0} - {1}", pair.Value, pair.Key);

}

Console.WriteLine("\nTable of numerical atoms:");

foreach (KeyValuePair<string, int> pair in numericAtomsTable)

{

Console.WriteLine("{0} - {1}", pair.Value, pair.Key);

}

Console.WriteLine("\nSyntactical vector:");

foreach (int i in sintVector)

{

Console.Write("{0} ", i);

}

Console.WriteLine("\n\nSemantic vector:");

foreach (int i in semtVector)

{

Console.Write("{0} ", i);

}

}