**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Г.ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и

автоматизированныхсистем

**КУРСОВОЙ ПРОЕКТ**

по дисциплине «Теория автоматов и формальных языков»

тема Обработка формальных языков

(наименование работы)

Автор работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (ФИО, группа)

Руководитель проекта \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Рязанов Ю.Д.

(подпись) (ФИО)

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Белгород

2021

ОГЛАВЛЕНИЕ

[**1.ВВЕДЕНИЕ**](#_pms3sibp1eu1) **……………………………………………………………………..3**

[**2.ОСНОВНАЯ ЧАСТЬ**](#_vnmrkk3hvnqk) **………………………………………………………….4**

[2.1.НИСХОДЯЩАЯ ОБРАБОТКА КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ](#_96s4e8gqndki) ………………………………………………………………………..4

[2.1.1.ПРЕОБРАЗОВАНИЕ ИСХОДНОЙ КС-ГРАММАТИКИ В LL(1) ГРАММАТИКУ](#_4yqk9lrf3rwp) ……………………………………………………………..5

[2.1.2.ОПРЕДЕЛЕНИЕ МНОЖЕСТВА ПЕРВЫХ](#_4ct867eexd3p) ………………………...6

[2.1.3.ОПРЕДЕЛЕНИЕ МНОЖЕСТВА СЛЕДУЮЩИХ](#_1cpm051uyvmb) ………………....7

[2.1.4.ОПРЕДЕЛЕНИЕ МНОЖЕСТВА ВЫБОРА](#_ewo34ng902si) ………………………...8

[2.1.5.ПОСТРОЕНИЕ НИСХОДЯЩЕГО МП-РАСПОЗНАВАТЕЛЯ ПО LL(1) ГРАММАТИКЕ](#_o8fo9m83wwk) ……………………………………………………...9

[2.1.6.РАЗРАБОТКА ТЕСТОВЫХ ДАННЫХ](#_9omutaadgiub) …………………………...11

[2.1.7.РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ](#_h77c56jto0bv) …………………………….14

[2.1.7.1.МЕТОД РЕКУРСИВНОГО СПУСКА](#_ljukyo60ksyr) ……………………….14

[2.1.7.2.НИСХОДЯЩИЙ МП РАСПОЗНАВАТЕЛЬ](#_xnb3sart1hph) ………………….17

[2.2.ВОСХОДЯЩАЯ ОБРАБОТКА КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ МЕТОДОМ «ПЕРЕНОС-ОПОЗНАНИЕ»](#_onmr2osjwi50) ……………………….20

[2.2.1.ОПРЕДЕЛЕНИЕ МНОЖЕСТВА ПЕРВЫХ И СЛЕДУЮЩИХ ДЛЯ КАЖДОГО СИМВОЛА ГРАММАТИКИ](#_s99o4n8xo8fr) ………………………....21

[2.2.2.ПОСТРОЕНИЕ УПРАВЛЯЮЩЕЙ ТАБЛИЦЫ ВОСХОДЯЩЕГО МП-РАСПОЗНАВАТЕЛЯ ТИПА «ПЕРЕНОС-ОПОЗНАНИЕ»](#_kgm6n0edhexd) ………..22

[2.2.3.ПОСТРОЕНИЕ ПРОЦЕДУРЫ ОПОЗНАНИЯ В ВИДЕ КОНЕЧНОГО АВТОМАТА](#_m5nvr45qw2q) ……………………………………………...23

[2.2.4.РАЗРАБОТКА ТЕСТОВЫХ ДАННЫХ](#_48hggtfuhka0) …………………………...24

[2.2.5.РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ](#_nbpg2ng24uz2) …………………………….27

[**3.АНАЛИЗ РАБОТЫ ПРОГРАММЫ**](#_wkgnrvotzcob) **………………………………………..29**

[3.1.АЛГОРИТМ РЕАЛИЗУЮЩИЙ НИСХОДЯЩИЙ МП РАСПОЗНАВАТЕЛЬ](#_l0da8cuiqq41) ………………………………………………………….29

[3.2.АЛГОРИТМ РЕКУРСИВНОГО СПУСКА](#_4xyh6z78p821) ……………………………..30

[3.3.АЛГОРИТМ “ПЕРЕНОС ОПОЗНАНИЕ”](#_wrjrd0jqsb0n) ……………………………....31

[**4.ЗАКЛЮЧЕНИЕ**](#_flwouce60o3f) **……………………………………………………………....32**

[**5.СПИСОК ЛИТЕРАТУРЫ**](#_1y810tw) **…………………………………………………...33**

[**6.ПРИЛОЖЕНИЕ**](#_4i7ojhp) **……………………………………………………………....34**

[6.1.Нисходящий МП автомат](#_wsp86ea7ttyl) ….……………………………………………..34

[6.2.Метод рекурсивного спуска](#_ksllunmehmla) ……………………………………………...37

[6.3.Алгоритм “перенос-опознание”](#_qssna5sajhwv) ………………………………………....41

# 1.ВВЕДЕНИЕ

Контекстно-свободные грамматики – частный случай формальной грамматики, в котором левые части правил состоят из одного нетерминала. Смысл термина «контекстно-свободная» заключается в том, что есть возможность применить продукцию к нетерминалу, причём независимо от контекста этого нетерминала. Язык, который может быть задан КС-грамматикой, называется контекстно-свободным языком или КС-языком.

КС-грамматики находят большое применение в информатике. Ими задается грамматическая структура большинства языков программирования, структурированных данных и т. д. Для разбора КС-грамматики достаточно автомата с магазинной памятью, для разбора других грамматик может потребоваться полная машина Тьюринга.

Существуют два разных класса распознавателей, их названия связаны с порядком построения дерева вывода. Как правило, все распознаватели читают входную цепочку символов слева направо, поскольку предполагается такая нотация в написании исходного текста программ:

- Нисходящие распознаватели, которые порождают цепочки левостороннего вывода и строят дерево вывода сверху вниз. Они используют модификации алгоритма с подбором альтернатив. При их создании применяется метод, который позволяет однозначно выбрать одну и только одну альтернативу на каждом шаге работы МП-автомата (шаг «выброс» в этом автомате всегда выполняется однозначно).

- Восходящие распознаватели, которые порождают цепочки правостороннего вывода и строят дерево вывода снизу вверх. Восходящие распознаватели используют модификации алгоритма «сдвиг-свертка» (или «перенос-свертка», что то же самое). При их создании применяются методы, которые позволяют однозначно выбрать между выполнением «сдвига» («переноса») или «свертки» на каждом шаге работы расширенного МП-автомата, а при выполнении свертки однозначно выбрать правило, по которому будет производиться свертка.

В данной работе целью стало изучение нисходящей обработки КС-языков, а также восходящего метода обработки формальных языков типа «перенос-опознание».

# 

# 2.ОСНОВНАЯ ЧАСТЬ

## 2.1.НИСХОДЯЩАЯ ОБРАБОТКА КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ

Для реализации нисходящей обработки контекстно-свободных языков была использована следующая грамматика

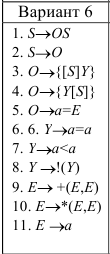


рис.1. Исходная грамматика 1

### 

### 2.1.1.ПРЕОБРАЗОВАНИЕ ИСХОДНОЙ КС-ГРАММАТИКИ В LL(1) ГРАММАТИКУ

Грамматику называют LL(1)-грамматикой, если для каждого нетерминала, появляющегося в левой части более одного порождающего правила, множества направляющих символов, соответствующих правым частям альтернативных порождающих правил – непересекающиеся. Все LL(1)-грамматики можно разбирать детерминировано сверху вниз.

Обычно очень большое число контекстно-свободных средств языка программирования можно представить с помощью LL(1)-грамматики. Проблема заключается в том, чтобы, имея грамматику, которая не обладает признаком LL(1), найти эквивалентную ей LL(1)-грамматику. Не существует универсального алгоритма преобразования любой КС-грамматики в LL(1) форму (а также определения самой возможности такого преобразования).

Однако существует ряд приемов, позволяющих выполнить такое преобразование во многих частных случаях.

· устранение левой рекурсии

· факторизация

Преобразуем данную грамматику G1 в G2, которая будет LL(1) грамматикой, проведем левую факторизацию.

G2

1. S -> OZ

2. Z -> S

3. Z -> e

4. O -> {P

5. P -> [S]Y}

6. P -> Y[S]}

7. O -> a=E

8. Y -> aW

9. W -> =a

10.W -> <a

11.Y -> !(Y)

12.E -> +(E,E)

13.E -> \*(E,E)

14.E -> a

### 2.1.2.ОПРЕДЕЛЕНИЕ МНОЖЕСТВА ПЕРВЫХ

Множество ПЕРВЫХ(FIRST) определяется следующим образом FIRST(α)={c∣α⇒∗cβ}∪{ε if α⇒∗ε}

Другими словами, FIRST(α) — все символы (терминалы), с которых могут начинаться всевозможные выводы из α

Найдем множество FIRST для заданной грамматики

FIRST(S) = { a

FIRST(Z) = { a e

FIRST(O) = { a

FIRST(P) = [ a !

FIRST(Y) = a !

FIRST(W) = < =

FIRST(E) = + \* a

### 

### 2.1.3.ОПРЕДЕЛЕНИЕ МНОЖЕСТВА СЛЕДУЮЩИХ

Множество СЛЕДУЮЩИХ(FOLLOW) определяется следующим образом FOLLOW(A)={c∣S⇒∗αAcβ}∪{-| if S⇒∗αA}

Другими словами FOLLOW(A) — всевозможные символы, которые встречаются после нетерминала A во всех не бесполезных правилах грамматики.

Найдем множество FOLLOW для заданной грамматики

FOLLOW(S) = -| ]

FOLLOW(Z) = -| ]

FOLLOW(O) = { a e

FOLLOW(P) = { a e

FOLLOW(Y) = ) [ }

FOLLOW(W) = ) [ }

FOLLOW(E) = { a e , )

### 

### 2.1.4.ОПРЕДЕЛЕНИЕ МНОЖЕСТВА ВЫБОРА

Множество выбор определяется следующим образом

Если:

1)A -> i и i не равно $

Выбор(A -> i) = Перв(i)

2)A -> $

Выбор(A -> $) = След(A)

3)A -> i и i равно $

Выбор(A -> i) = Перв(i) & След(A) & {$}

Контекстно-свободная грамматика, называется LL(1) грамматикой, если множество ВЫБОРА(SELECT) построено для правил с одинаковым левым нетерминалом не содержит одинаковых элементов.

Построение множества SELECT необходимо не только для определения является ли грамматика LL(1) грамматикой, но также и для создания нисходящих распознавателей для них.

Определим множество выбора для заданной грамматики

1. SELECT(S -> OZ) = { a

2. SELECT(Z -> S) = { a

3. SELECT(Z -> e) = -| ]

4. SELECT(O -> {P ) = {

5. SELECT(P -> [S]Y}) = [

6. SELECT(P -> Y[S]}) = a !

7. SELECT(O -> a=E) = a

8. SELECT(Y -> aW ) = a

9. SELECT(W -> =a) = =

10.SELECT(W -> <a) = <

11.SELECT(Y -> !(Y)) = !

12.SELECT(E -> +(E,E) = +

13.SELECT(E -> \*(E,E)) = \*

14.SELECT(E -> a) = a

Множества выбора у правил с одинаковой левой частью не пересекаются это значит что грамматика и правда LL(1)

### 

### 2.1.5.ПОСТРОЕНИЕ НИСХОДЯЩЕГО МП-РАСПОЗНАВАТЕЛЯ ПО LL(1) ГРАММАТИКЕ

Таблица для нисходящего МП распознавателя строится по множеству выбора: когда на вершине стека нетерминал A, он замещается на правую часть правила A->a, если текущий символ строки b принадлежит множеству SELECT(A->a), текущий символ при этом заменяется на следующий.

В случае если текущий символ и верхний магазинный символ одинаковы осуществляется выброс.

Начальное состояние магазина: VS

Цепочка считается допущенной, если магазин пустой и вся цепочка обработана

Построим управляющую таблицу для заданной грамматики:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | { | ] | [ | a | = | < | ! | + | \* | -| |
| S | S->OZ |  |  | S -> OZ |  |  |  |  |  |  |
| Z | Z->S | Z->e |  | Z -> S |  |  |  |  |  | Z->e |
| O | O->{P |  |  | O ->a=E |  |  |  |  |  |  |
| P |  |  | P->[S]Y} | P->Y[S]} |  |  | P->Y[S]} |  |  |  |
| Y |  |  |  | Y -> aW |  |  | Y->!(Y) |  |  |  |
| W |  |  |  |  | W->=a | W-><a |  |  |  |  |
| E |  |  |  | E -> a |  |  |  | E->+(E,E) | E->\*(E,E) |  |
| { | Выброс |  |  |  |  |  |  |  |  |  |
| ] |  | Выброс |  |  |  |  |  |  |  |  |
| [ |  |  | Выброс |  |  |  |  |  |  |  |
| a |  |  |  | Выброс |  |  |  |  |  |  |
| = |  |  |  |  | Выброс |  |  |  |  |  |
| < |  |  |  |  |  | Выброс |  |  |  |  |
| ! |  |  |  |  |  |  | Выброс |  |  |  |
| + |  |  |  |  |  |  |  | Выброс |  |  |
| \* |  |  |  |  |  |  |  |  | Выброс |  |
| <> |  |  |  |  |  |  |  |  |  | Допустить |

таблица.1. управляющая таблица нисходящего МП автомата

### 

### 2.1.6.РАЗРАБОТКА ТЕСТОВЫХ ДАННЫХ

Принадлежащие языку

{[a=+(\*(a,a),a)]a<a}

Левый вывод

S => OZ => {PZ => {[S]Y}Z => {[a=EZ]Y}Z => {[a=+(E,E)Z]Y}Z => {[a=+(\*(E,E),E)Z]Y}Z => {[a=+(\*(a,E),E)Z]Y}Z => {[a=+(\*(a,a),E)Z]Y}Z => {[a=+(\*(a,a),a)Z]Y}Z => {[a=+(\*(a,a),a)Y]Y}Z => {[a=+(\*(a,a),a)aW]Y}Z => {[a=+(\*(a,a),a)a<a]Y}Z => {[a=+(\*(a,a),a)a<a]Y}

Дерево вывода

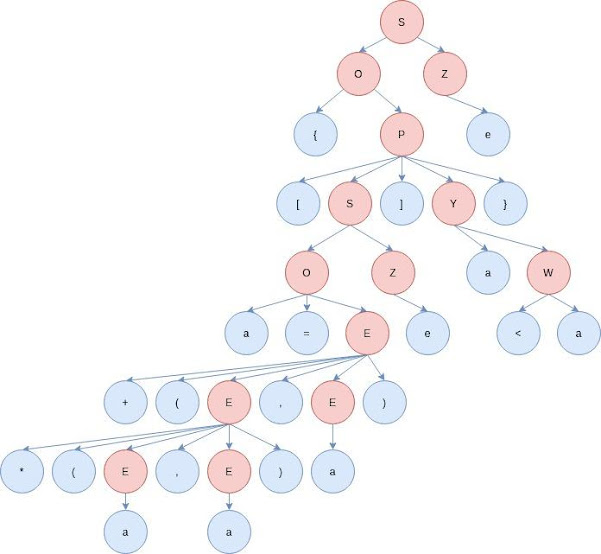


рис.2.1.1.Дерево вывода допустимой цепочки

a=a{a=a[{!(a<a)[a=a]}]}

Левый вывод

S => OZ => a=EZ => a=aZ => a=aS => a=aOZ => a=a{PZ => a=a{Y[S]}Z => a=a{aW[S]}Z => a=a{a=a[S]}Z => a=a{a=a[OS]}Z => a=a{a=a[{PZ]}Z => a=a{a=a[{Y[S]}Z]}Z => a=a{a=a[{!(Y)[S]}Z]}Z => a=a{a=a[{!(aW)[S]}Z]}Z => a=a{a=a[{!(a<a)[S]}Z]}Z => a=a{a=a[{!(a<a)[OZ]}Z]}Z => a=a{a=a[{!(a<a)[a=EZ]}Z]}Z => a=a{a=a[{!(a<a)[a=aZ]}Z]}Z => a=a{a=a[{!(a<a)[a=a]}Z]}Z => a=a{a=a[{!(a<a)[a=a]}]}Z => a=a{a=a[{!(a<a)[a=a]}]}

Дерево вывода

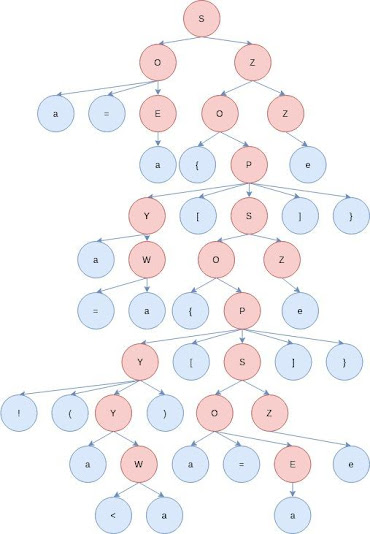


рис.2.1.2.Дерево вывода допустимой цепочки

Не принадлежащие языку

a

{a=+(\*(a,a),a)]}

{[}

### 

### 2.1.7.РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

#### 2.1.7.1.МЕТОД РЕКУРСИВНОГО СПУСКА

Проверим рекурсивный спуск

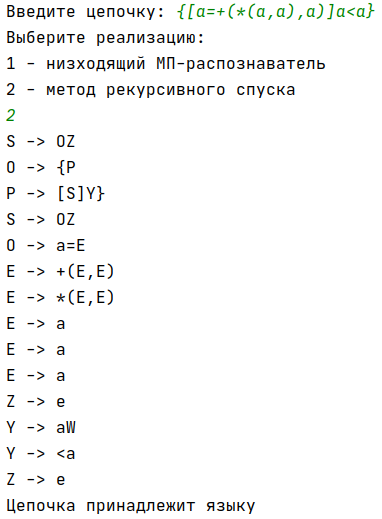


рис.2.1.3. Пример работы программы рекурсивного спуска

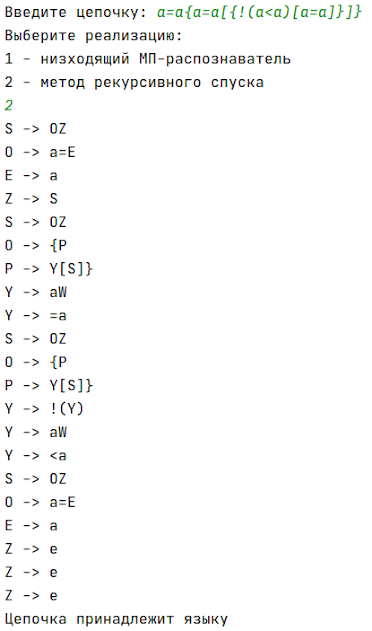


рис.2.1.4. Пример работы программы рекурсивного спуска

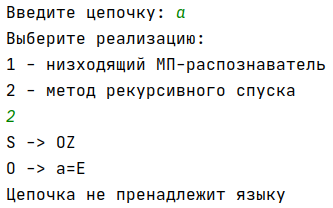


рис.2.1.5. Пример работы программы рекурсивного спуска

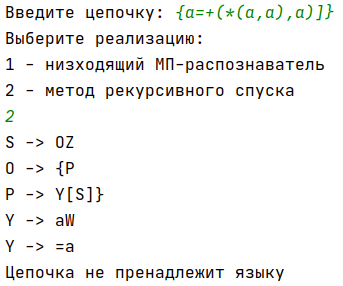


рис.2.1.6. Пример работы программы рекурсивного спуска

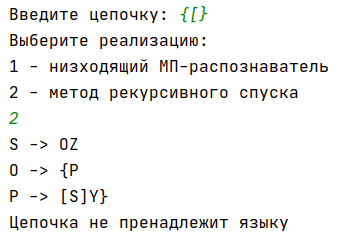


рис.2.1.7. Пример работы программы рекурсивного спуска

#### 

#### 2.1.7.2.НИСХОДЯЩИЙ МП РАСПОЗНАВАТЕЛЬ

Проверим нисходящий МП распознователь

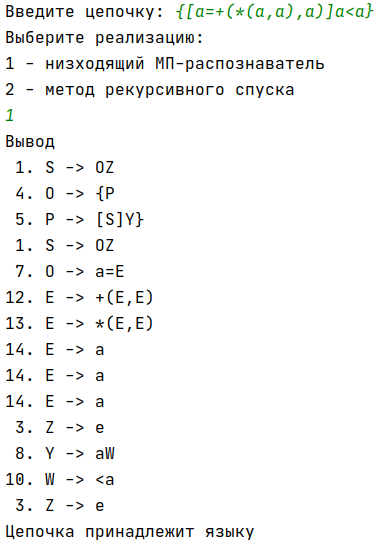


рис.2.1.8. Пример работы программы нисходящего МП автомата

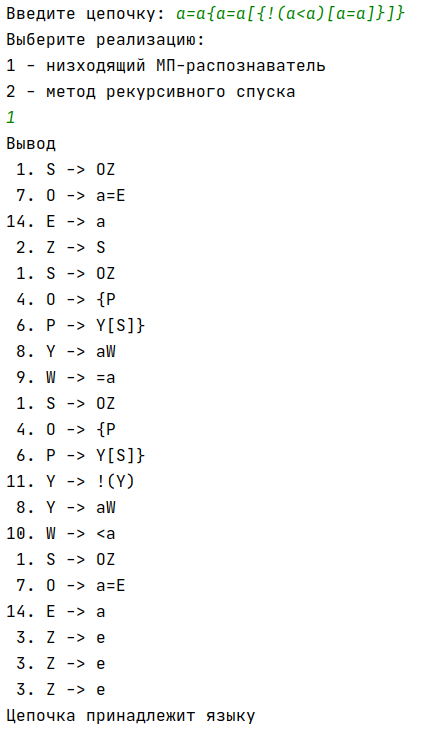


рис.2.1.9. Пример работы программы нисходящего МП автомата

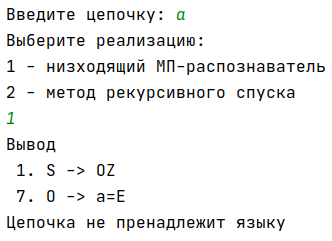


рис.2.1.10. Пример работы программы нисходящего МП автомата

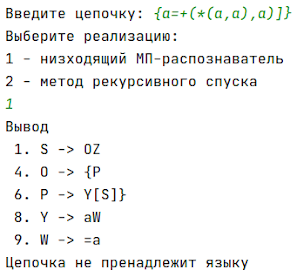


рис.2.1.11. Пример работы программы нисходящего МП автомата

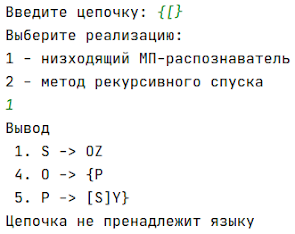


рис.2.1.12. Пример работы программы нисходящего МП автомата

## 

## 2.2.ВОСХОДЯЩАЯ ОБРАБОТКА КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ МЕТОДОМ «ПЕРЕНОС-ОПОЗНАНИЕ»

Для реализации метода «перенос-опознание» была использована следующая грамматика:

### 

рис.2.2.1. Исходная грамматика 2

В данной КС-грамматике нет правой рекурсии, поэтому дополнительных преобразований не потребовалось.

### 2.2.1.ОПРЕДЕЛЕНИЕ МНОЖЕСТВА ПЕРВЫХ И СЛЕДУЮЩИХ ДЛЯ КАЖДОГО СИМВОЛА ГРАММАТИКИ

Для определения множества FIRST и FOLLOW, представленных в таблице используется следующий алгоритм, он схож с тем что используется при нисходящей обработке, за исключением того что в множестве FIRST присутствуют нетерминалы.

Изначально просматриваются все правила грамматики. В множество FIRST заносятся все те символы, с которых начинаются правые части правил, а также первые символы для нетерминалов, входящих в это множество.

В множество FOLLOW записываются все терминалы, следующее за символом в правых частях правил. Если следующий символ оказывается нетерминалом, но необходимо добавить терминалы, которые входят в множество первых для него. Иначе, если символ – самый правый в каком-либо правиле, то множество следующих нужно включить в множество следующих для нетерминала в левой части правил.

В таблицах, представленных ниже, существуют следующие соответствия: концевой маркер обозначен знаком -|, а маркер дна – символом V. Данные обозначения будут также использованы при написании программы по методу «перенос-опознание».

Пустые ячейки в таблице распознавателя являются состоянием ошибки.

Найдем множество FIRST

FIRST(S) = S O { a

FIRST(O) = O { a

FIRST(Y) = Y | & ! a

FIRST(E) = E a

Найдем множество FOLLOW

FOLLOW(S) = -| ; ]

FOLLOW(O) = -| ; ]

FOLLOW(Y) = [ , )

FOLLOW(E) = -| ; ] + \* a

### 

### 2.2.2.ПОСТРОЕНИЕ УПРАВЛЯЮЩЕЙ ТАБЛИЦЫ ВОСХОДЯЩЕГО МП-РАСПОЗНАВАТЕЛЯ ТИПА «ПЕРЕНОС-ОПОЗНАНИЕ»

Для построения управляющей таблицы восходящего МП-распознавателя типа «перенос-опознание» необходимо использовать следующие правила:

1. 𝑇[𝑚,𝑥]⟵П,если ∃𝐴→𝛼𝑚𝛽 и 𝑥∈FIRST(𝛽) или 𝑚=𝑉 и 𝑥∈FIRST(𝑆)
2. 𝑇[𝑚,𝑥]⟵ОП,если ∃𝐴→𝛼𝑚 и 𝑥∈FOLLOW(А) или 𝑚=𝑆 и 𝑥=−|

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| НАД |  | { | [ | ] | } | a | = | ( | | | ) | ! | & | + | \* | ; | , | -| |
|  | S |  |  | П |  |  |  |  |  |  |  |  |  |  | П |  | ОП |
|  | O |  |  | ОП |  |  |  |  |  |  |  |  |  |  | ОП |  | ОП |
|  | Y |  | П |  |  |  |  |  |  | П |  |  |  |  |  | П |  |
| E | E |  |  | ОП |  | П |  |  |  |  |  |  | П | П | ОП |  | ОП |
| Y | { |  |  |  |  | П |  |  | П |  | П | П |  |  |  |  |  |
| S O | [ | П |  |  |  | П |  |  |  |  |  |  |  |  |  |  |  |
|  | ] |  |  |  | П |  |  |  |  |  |  |  |  |  |  |  |  |
|  | } |  |  | ОП |  |  |  |  |  |  |  |  |  |  | ОП |  | ОП |
|  | a |  | ОП | ОП |  | ОП | П |  |  | ОП |  |  | ОП | ОП | ОП | ОП | ОП |
| E | = |  |  |  |  | П |  |  |  |  |  |  |  |  |  |  |  |
| Y | ( |  |  |  |  | П |  |  | П |  | П | П |  |  |  |  |  |
|  | | |  |  |  |  |  |  | П |  |  |  |  |  |  |  |  |  |
|  | ) |  | ОП |  |  |  |  |  |  | ОП |  |  |  |  |  | ОП |  |
|  | ! |  |  |  |  |  |  | П |  |  |  |  |  |  |  |  |  |
|  | & |  |  |  |  |  |  | П |  |  |  |  |  |  |  |  |  |
|  | + |  |  | ОП |  | ОП |  |  |  |  |  |  | ОП | ОП | ОП |  | ОП |
|  | \* |  |  | ОП |  | ОП |  |  |  |  |  |  | ОП | ОП | ОП |  | ОП |
| O | ; | П |  |  |  | П |  |  |  |  |  |  |  |  |  |  |  |
| Y | , |  |  |  |  | П |  |  | П |  | П | П |  |  |  |  |  |
| S O | V | П |  |  |  | П |  |  |  |  |  |  |  |  |  |  |  |

таблица 2 управляющая таблица восходящего МП автомата

### 

### 2.2.3.ПОСТРОЕНИЕ ПРОЦЕДУРЫ ОПОЗНАНИЯ В ВИДЕ КОНЕЧНОГО АВТОМАТА

Для построения автомата необходимо последовательно просматривать правые части грамматики справа налево и по символам грамматики производить переходы в следующие узлы графа.

Конфликты «опознание-опознание» можно решать с помощью множеств ПОД и СЛЕДУЮЩИХ.

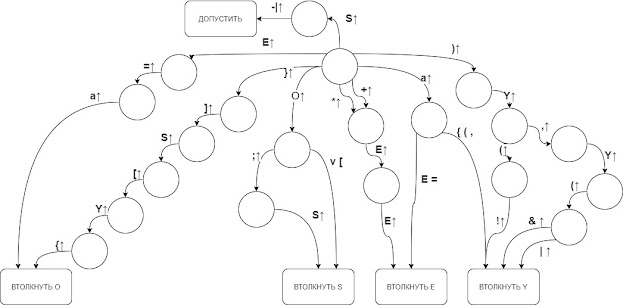


рис.2.2.2. Процедура Опозания

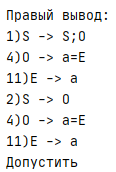
### 

### 2.2.4.РАЗРАБОТКА ТЕСТОВЫХ ДАННЫХ

Допустимые цепочки

a=a;a=a

Правый вывод



Дерево вывода

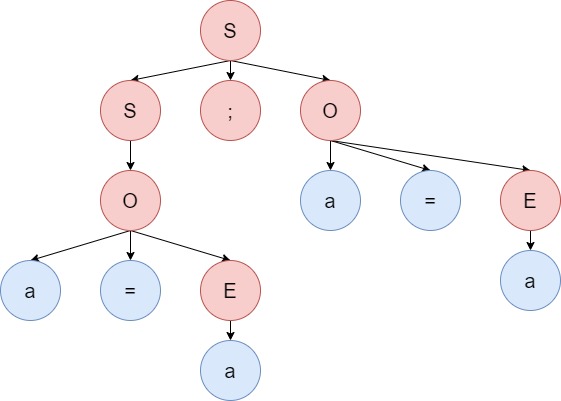
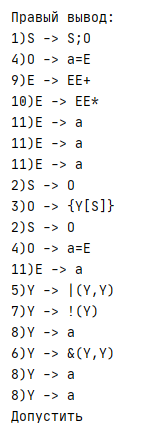


рис.2.2.3. Дерево вывода допускаемой цепочки

{|(&(a,a),!(a))[a=a]};a=aaa\*+

Правый вывод



Дерево вывода

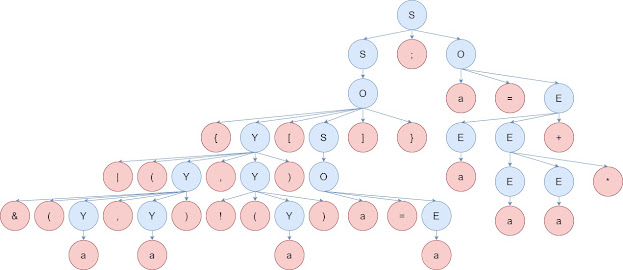


рис.2.2.4. Дерево вывода допускаемой цепочки

Недопустимые

{|(&(a,a),!(a))

+{|(&(a,a),!(a))[a=a]};a=aaa

### 

### 2.2.5.РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

Допустимые

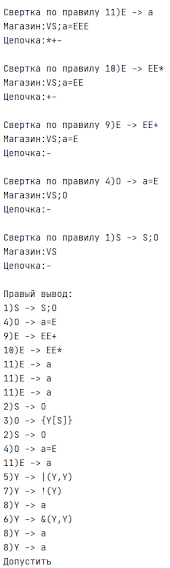
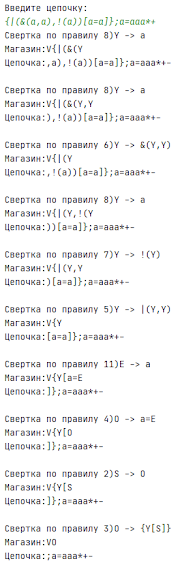
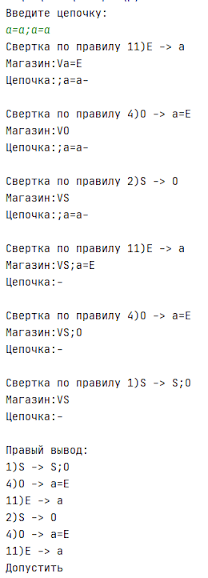


рис.2.2.5. Пример работы программы с допустимыми цепочками

Недопустимые

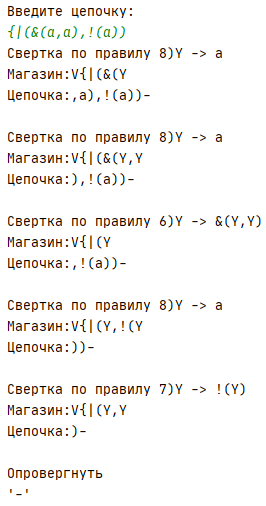


рис.2.2.6. Пример работы программы с недопустимыми цепочками

### 

# 

# 3.АНАЛИЗ РАБОТЫ ПРОГРАММЫ

## 3.1.АЛГОРИТМ РЕАЛИЗУЮЩИЙ НИСХОДЯЩИЙ МП РАСПОЗНАВАТЕЛЬ

В данном алгоритме использовался автомат с магазинной памятью. Базовыми структурами данных являются две строки:

G\_NOT\_TERMINALS = 'SZOPYWE'

G\_TERMINALS = '{[a=<!+\*]}(,)'

Содержащие нетерминалы и терминалы соответственно, а также список правил грамматики:

RULES = [('S', 'OZ'),

('Z', 'S'),

('Z', ''),

('O', '{P'),

('P', '[S]Y}'),

('P', 'Y[S]}'),

('O', 'a=E'),

('Y', 'aW'),

('W', '=a'),

('W', '<a'),

('Y', '!(Y)'),

('E', '+(E,E)'),

('E', '\*(E,E)'),

('E', 'a')]

И управляющая таблица МП автомата TABLE

Алгоритм зависит от длины поступающей цепочки и имеет сложность O(N)

Объем программы составил: 2.2Кб

## 

## 3.2.АЛГОРИТМ РЕКУРСИВНОГО СПУСКА

Данный алгоритм также является алгоритмом нисходящей обработки цепочки строящий левый вывод. Только за место явного использования стека и автомата, программа представляет из себя набор функций соответствующих нетерминалам которые вызывают друг друга. Явно используемый стек при обработки МП распознавателем заменяется на неявное использование стека вызовов функций.

Основными структурами данных также являются две строки:

G\_NOT\_TERMINALS = 'SZOPYWE'

G\_TERMINALS = '{[a=<!+\*]}(,)'

Управляющей таблицы нет, так как то какая функция будет вызвана определяет текущая функция.

Объем программы составил: 3.4Кб

## 

## 3.3.АЛГОРИТМ “ПЕРЕНОС ОПОЗНАНИЕ”

В данной реализации алгоритма реализуется процедура опознания путем сравнения верха автомата с правыми частями правил. Так же используется множество ПОД для определения того можно ли провести замену

Основными структурами данных являются G представляющая из себя список правил грамматики:

G = [("S", "S;O"),

("S", "O"),

("O", "{Y[S]}"),

("O", "a=E"),

("Y", "|(Y,Y)"),

("Y", "&(Y,Y)"),

("Y", "!(Y)"),

("Y", "a"),

("E", "EE+"),

("E", "EE\*"),

("E", "a"), ]

Множество P (ПОД) для всех нетерминалов:

P = {

"S": "[V",

"O": "[;V",

"Y": "{(,",

"E": "E=",

}

Скорость выполнения программы зависит от длины цепочки. Следовательно, порядок функции временной сложности алгоритма O(N), где N – длина цепочки.

Объем программы составил: 3.6Кб

# 

# 4.ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были изучены как нисходящие, так и восходящие алгоритмы. Для данных алгоритмов были использованы две различные грамматики, на основе которых и проводилась разработка программ.

Перед тем, как приступить к кодированию, была проведена работа с грамматиками. Для метода «перенос-опознание» были найдены множества ПЕРВЫХ и СЛЕДУЮЩИХ, была построена управляющая таблица восходящего МП-распознавателя данного типа, а также произведено отображение процедуры опознания в виде конечного автомата. Для нисходящих методов были найдены множества ПЕРВЫХ, СЛЕДУЮЩИХ и ВЫБОРА. Построена управляющая таблица автомата. Также была проведена левая факторизация для удаления правого ветвления.

Следующим этапом стало создание программ. В ходе разработки программ были закреплены навыки по работе в среде программирования PyCharm, а также получен опыт программирования на языке Python.

Итогом работы были получены три программы сильно отличающиеся в реализации, но одинаково справляющиеся с поставленной целью.

Метод “перенос-опознание” менее требователен к исходной грамматике, но при построении распознавателя могут возникнуть конфликты, для решения которых все таки будет необходимо преобразовывать грамматику и находить множество “ПОД”. Также он требует большего объема памяти, чем другие методы.

Нисходящий распознаватель более эффективен по памяти и времени распознавания, однако для его применения требуется LL(1) грамматика, получение которой из исходной не детерминировано, в результате чего могут возникнуть трудности с преобразованием. Таким образом, применимость нисходящего МП-распознавателя снижается.

Метод рекурсивного спуска наиболее эффективен по памяти из представленных и не требует построения таблицы распознавателя, однако для его построения так же как и для нисходящего распознавателя требуется LL(1) грамматика. Минусом данного метода также является сложность модифицирования, написанную реализацию невозможно изменить для распознавания другой грамматики.

Таким образом, все три метода представляют интерес при обработки формальных языков.

# 5.СПИСОК ЛИТЕРАТУРЫ

1. А. Ахо, Дж. Ульман. Компиляторы, принципы, технологии и инструменты. 2008 г.

2. Никлаус Вирт Построение компиляторов / Пер. с англ. Борисо. Е. В., Чернышов Л. Н. - М.: ДМК Пресс, 2010. - 192 с.: ил.

3. Хопкрофт Д., Мотвани Р., Ульман Д. — Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. — Москва, Издательский дом «Вильямс», 2002. — 528 с. : ISBN 5-8459-0261-4 (рус.)

4. Университет ИТМО [Электронный ресурс] URL: https://neerc.ifmo.ru/wiki /index/php?title= Контекстно-свободные грамматики, вывод, лево- и правосторонний вывод, дерево разбора (дата обращения 20.05.2019)

5. Wikipedia [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Контекстно-свободная\_грамматика (дата обращения 19.05.2019).

# 6.ПРИЛОЖЕНИЕ

## 6.1.Нисходящий МП автомат

G\_NOT\_TERMINALS = 'SZOPYWE'

G\_TERMINALS = '{[a=<!+\*]}(,)'

RULES = [('S', 'OZ'),

('Z', 'S'),

('Z', ''),

('O', '{P'),

('P', '[S]Y}'),

('P', 'Y[S]}'),

('O', 'a=E'),

('Y', 'aW'),

('W', '=a'),

('W', '<a'),

('Y', '!(Y)'),

('E', '+(E,E)'),

('E', '\*(E,E)'),

('E', 'a')]

class MP:

class ParseError(Exception):

def \_\_init\_\_(self):

super().\_\_init\_\_('Цепочка не пренадлежит языку')

TABLE = {

'S': {

'{': RULES[0],

'a': RULES[0]

},

'Z': {

'{': RULES[1],

'a': RULES[1],

'$': RULES[2],

']': RULES[2]

},

'O': {

'{': RULES[3],

'a': RULES[6]

},

'P': {

'[': RULES[4],

'a': RULES[5],

'!': RULES[5]

},

'Y': {

'a': RULES[7],

'!': RULES[10]

},

'W': {

'=': RULES[8],

'<': RULES[9]

},

'E': {

'a': RULES[13],

'+': RULES[11],

'\*': RULES[12]

}

}

def \_print\_rule(self, rule: (str, str)):

rule\_num = RULES.index(rule) + 1

print(f'{" " if rule\_num < 10 else ""}{rule\_num}. {rule[0]} -> {rule[1] if rule[1] else "e"}')

def check(self, chain: str):

stack = ['$', 'S']

chain += '$'

indexInSym = 0

run\_flag = True

print('Вывод')

while run\_flag:

X = stack[-1]

if (X in G\_TERMINALS) or (X == '$'):

if X == chain[indexInSym]:

stack.pop()

indexInSym += 1

else:

raise MP.ParseError()

else:

try:

rule: (str, str) = MP.TABLE[X][chain[indexInSym]] # тут вылетит ошибка если ячейка таблицы пуста

stack.pop()

stack += rule[1][::-1]

self.\_print\_rule(rule)

except:

raise MP.ParseError()

run\_flag = (X != '$')

## 

## 6.2.Метод рекурсивного спуска

G\_NOT\_TERMINALS = 'SZOPYWE'

G\_TERMINALS = '{[a=<!+\*]}(,)'

class Recurse:

def \_\_init\_\_(self, chain: str):

self.chain = chain

def S(self):

FIRST = {

'{': 'OZ',

'a': 'OZ',

}

try:

inSym = self.chain[0]

print(f'S -> {FIRST[inSym]}')

self.parse(FIRST[inSym])

except:

raise MP.ParseError() # В данном случае из нетерминала не выводится пустая цепочка

def Z(self):

FIRST = {

'{': 'S',

'a': 'S'

}

try:

inSym = self.chain[0]

print(f'Z -> {FIRST[inSym]}')

self.parse(FIRST[inSym])

except:

if self.chain[0] in ']$':

print(f'Z -> e')

else:

raise MP.ParseError() # В данном случае из нетерминала не выводится пустая цепочка

def O(self):

FIRST = {

'{': '{P',

'a': 'a=E'

}

try:

inSym = self.chain[0]

print(f'O -> {FIRST[inSym]}')

self.parse(FIRST[inSym])

except:

raise MP.ParseError() # В данном случае из нетерминала не выводится пустая цепочка

def P(self):

FIRST = {

'[': '[S]Y}',

'a': 'Y[S]}',

'!': 'Y[S]}'

}

try:

inSym = self.chain[0]

print(f'P -> {FIRST[inSym]}')

self.parse(FIRST[inSym])

except:

raise MP.ParseError() # В данном случае из нетерминала не выводится пустая цепочка

def Y(self):

FIRST = {

'a': 'aW',

'!': '!(Y)'

}

try:

inSym = self.chain[0]

print(f'Y -> {FIRST[inSym]}')

self.parse(FIRST[inSym])

except:

raise MP.ParseError() # В данном случае из нетерминала не выводится пустая цепочка

def W(self):

FIRST = {

'=': '=a',

'<': '<a'

}

try:

inSym = self.chain[0]

print(f'Y -> {FIRST[inSym]}')

self.parse(FIRST[inSym])

except:

raise MP.ParseError() # В данном случае из нетерминала не выводится пустая цепочка

def E(self):

FIRST = {

'a': 'a',

'+': '+(E,E)',

'\*': '\*(E,E)'

}

try:

inSym = self.chain[0]

print(f'E -> {FIRST[inSym]}')

self.parse(FIRST[inSym])

except:

raise MP.ParseError() # В данном случае из нетерминала не выводится пустая цепочка

def parse(self, u:str):

v = u

while v:

X = v[0]

z = v[1::]

if X in G\_TERMINALS:

if X != self.chain[0]:

raise MP.ParseError()

else:

self.chain = self.chain[1::]

else:

{

'S': self.S,

'Z': self.Z,

'O': self.O,

'P': self.P,

'Y': self.Y,

'W': self.W,

'E': self.E,

}[X]()

v = z

## 

## 6.3.Алгоритм “перенос-опознание”

G = [("S", "S;O"),

("S", "O"),

("O", "{Y[S]}"),

("O", "a=E"),

("Y", "|(Y,Y)"),

("Y", "&(Y,Y)"),

("Y", "!(Y)"),

("Y", "a"),

("E", "EE+"),

("E", "EE\*"),

("E", "a"), ]

T = {

"S": {

"]": "П",

";": "П",

"-": "П",

},

"O": {

"]": "ОП",

";": "ОП",

"-": "ОП",

},

"Y": {

"[": "П",

")": "П",

",": "П",

},

"E": {

"-": "ОП",

";": "ОП",

"\*": "П",

"+": "П",

"a": "П",

"]": "ОП",

},

"{": {

"&": "П",

"!": "П",

"|": "П",

"a": "П",

},

"[": {

"a": "П",

"{": "П",

},

"]": {

"}": "П",

},

"}": {

"-": "ОП",

";": "ОП",

"]": "ОП",

},

"a": {

"-": "ОП",

",": "ОП",

";": "ОП",

"\*": "ОП",

"+": "ОП",

")": "ОП",

"=": "П",

"a": "ОП",

"]": "ОП",

"[": "ОП",

},

"=": {

"a": "П",

},

"(": {

"&": "П",

"!": "П",

"|": "П",

"a": "П",

},

"|": {

"(": "П",

},

")": {

",": "ОП",

")": "ОП",

"[": "ОП",

},

"!": {

"(": "П",

},

"&": {

"(": "П",

},

"+": {

"-": "ОП",

";": "ОП",

"\*": "ОП",

"+": "ОП",

"a": "ОП",

"]": "ОП",

},

"\*": {

"-": "ОП",

";": "ОП",

"\*": "ОП",

"+": "ОП",

"a": "ОП",

"]": "ОП",

},

";": {

"a": "П",

"{": "П",

},

",": {

"&": "П",

"!": "П",

"|": "П",

"a": "П",

},

"V": {

"a": "П",

"{": "П",

},

}

P = {

"S": "[V",

"O": "[;V",

"Y": "{(,",

"E": "E=",

}

class UpstreamRecognizer:

def \_\_init\_\_(self, \_G, \_T, \_P):

self.G = \_G

self.T = \_T

self.P = \_P

def recognize(self, string: str):

string += "-"

stack = "V"

rule\_seq = []

while string:

t\_row = self.T[stack[-1]]

t\_value = t\_row[string[0]]

if t\_value == "П":

stack += string[0]

string = string[1:]

elif t\_value == "ОП":

if stack[-1] == "S" and string == "-":

break

for index\_rule, rule in enumerate(self.G):

len\_stack\_without\_rule = len(stack) - len(rule[1])

if rule[1] == stack[len\_stack\_without\_rule:] and \

stack[len\_stack\_without\_rule-1] in self.P[rule[0]]:

stack = stack[:len\_stack\_without\_rule] + rule[0]

rule\_seq.append(f"{index\_rule+1}){rule[0]} -> {rule[1]}")

print(f"Свертка по правилу {index\_rule+1}){rule[0]} -> {rule[1]}\nМагазин:{stack}\nЦепочка:{string}\n")

break

else:

raise self.RecognizeException(f"Ошибка:\nМагазине: {stack}\nЦепочка: {string}\n")

rule\_seq.reverse()

print("Правый вывод:")

for rule in rule\_seq:

print(rule)

print("Допустить")

class RecognizeException(Exception):

pass

if \_\_name\_\_ == "\_\_main\_\_":

print("Введите цепочку: ")

string = input()

recognizer = UpstreamRecognizer(G, T, P)

try:

recognizer.recognize(string)

except Exception as e:

print("Опровергнуть")

print(e)