Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Кафедра «Информатика»
кафедра

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ

Лабораторная работа №5. Синтаксический анализ контекстно-свободных языков (часть 2)
Тема

Преподаватель _____ Д. В. Личаргин
подпись, дата | инициалы, фамилия

Студент КИ19-16/1б 031939175 _____ А. Д. Непомнящий
номер группы, зачетной книжки | подпись, дата | инициалы, фамилия

Красноярск 2021

1 Цель работы

Исследование контекстно-свободных грамматик и алгоритмов синтаксического анализа контекстно-свободных языков.

2 Задачи

Задачи работы разделены на три части и состоят в следующем.

Часть 1. Необходимо с использованием системы JFLAP, построить LL(1)-грамматику, описывающую заданный язык, или формально доказать невозможность этого. Полученная грамматика не должна повторять SLR(1)- грамматику, конструируемую в части 3.

Часть 2. Предложить программную реализацию метода рекурсивного спуска для распознавания строк заданного языка. Представить формальное доказательство принадлежности к классу LL(1) грамматики, лежащей в основе синтаксического анализа заданного языка. Во всех случаях язык должен состоять из последовательностей выражений. В качестве разделителя может выступать символ новой строки, точка с запятой или любой другой символ, не задействованный в других лексемах. Результатом работы синтаксического анализатора является выдача сообщения «Accepted» или «Rejected».

Часть 3. Необходимо с использованием системы JFLAP, построить SLR(1)-грамматику, описывающую заданный язык, или формально доказать невозможность этого. Во всех случаях реализуется язык, состоящий из последовательностей операторов присваивания. В качестве разделителя может выступать символ новой строки, точка с запятой или любой другой символ, не задействованный в прочих лексемах. В качестве L-значения оператора присваивания выступает только имя переменной. В правой части оператора присваивания указывается выражение, элементы которых оговариваются в каждом варианте задания. Полученная грамматика не должна повторять LL(1)-грамматику, конструируемую в части 1.

Часть 1: Язык оператора присваивания, в правой части которого задано арифметическое выражение. Элементами выражений являются целочисленные константы в двоичной системе счисления, имена переменных из одного символа

(от a до f), знаки операций и скобки для изменения порядка вычисления подвыражений. Операции (в сторону уменьшения приоритета): унарный минус, мультипликативные, аддитивные, присваивание.

Часть 2: Язык арифметических выражений, элементами которых являются целочисленные константы в двоичной, восьмеричной или десятичной системах счисления, имена переменных из 1-2 символов, знаки операций и скобки для изменения порядка вычисления подвыражений. Операции (в сторону уменьшения приоритета): унарный минус, мультипликативные, аддитивные, присваивание.

Часть 3: Элементами арифметического выражения являются целочисленные константы в 2- и 10-чной системах счисления, имена переменных из одного символа (от a до f), знаки операций и скобки для изменения порядка вычисления подвыражений. Операции (в сторону уменьшения приоритета): унарный минус, мультипликативные, аддитивные, присваивание.

3 Ход работы

3.1 Построение LL(1)-грамматики

Была реализована LL(1)-грамматика с помощью системы JFLAP. Она приведена на рисунке 1. На рисунке 2 приведена таблица синтаксического LL(1)-анализа, на рисунке 3 – результат анализа некоторых строк, на рисунках 4 и 5 – подробный процесс синтаксического анализа одной из принимаемых строк и одной из не принимаемых соответственно.

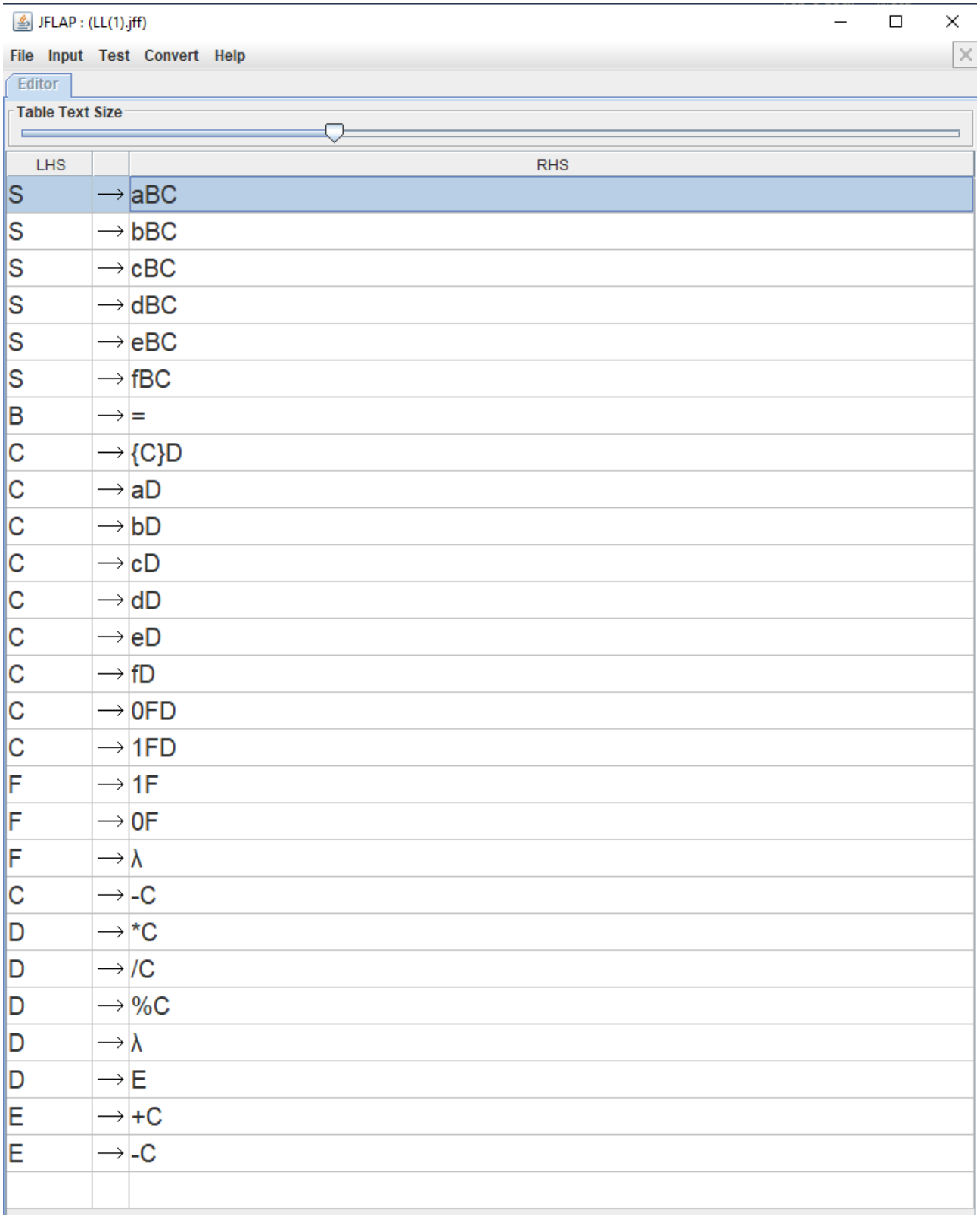

Рисунок 1 – полученная LL(1)-грамматика

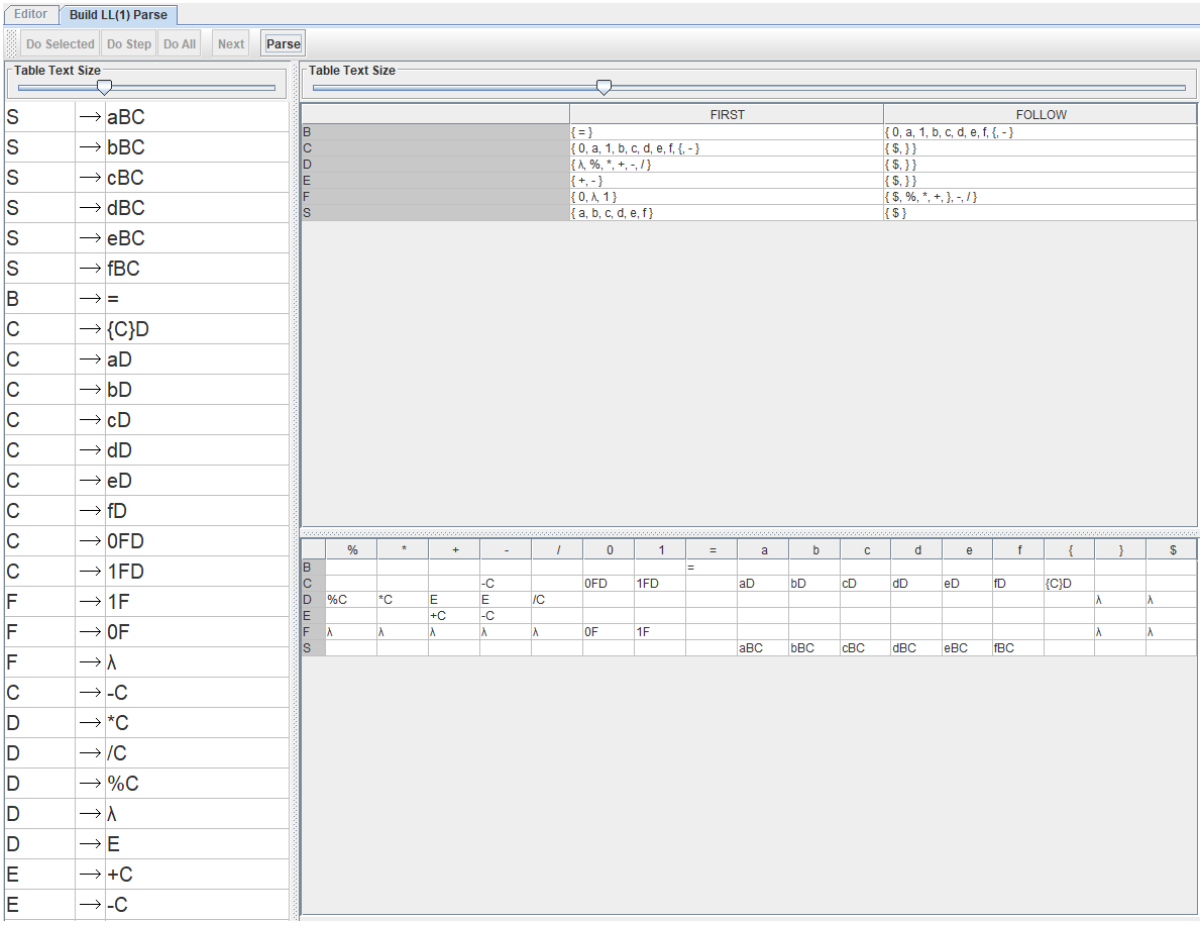

Рисунок 2 – таблица синтаксического LL(1)-анализа

Input	Result
a=a*b	Accept
b={a*1010}	Accept
c=a*111111111111111*a	Accept
d={b*1}*c	Accept
e=a*{d*e}*f	Accept
f=a*{a}*a	Accept
a={a*-1010}	Accept
b=a*-111111111111111*a	Accept
c={b*1}*c	Accept
d=1+{a-{b*c}/f}	Accept
e=a*-{d*e}*f	Accept
f=a*{{-a}}*a	Accept
a=a--b	Accept
reject_below	Reject
b=a**b	Reject
c=abc	Reject
d=a*-{{de}*f	Reject
e=a*{-a}}*a	Reject
a+b	Reject

Рисунок 3 – Результат анализа некоторых строк

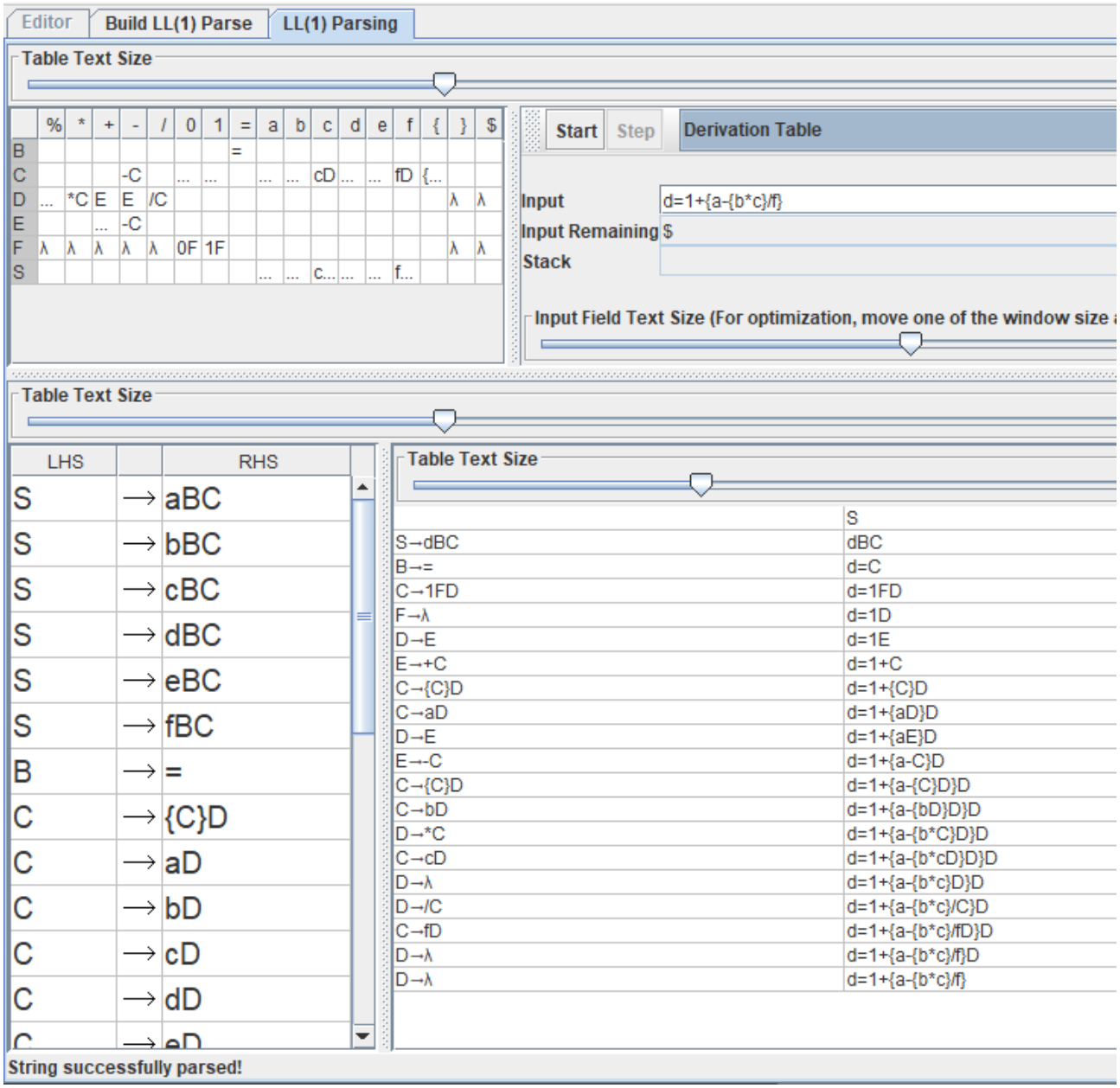

Рисунок 4 – Синтаксический анализ одной из строк

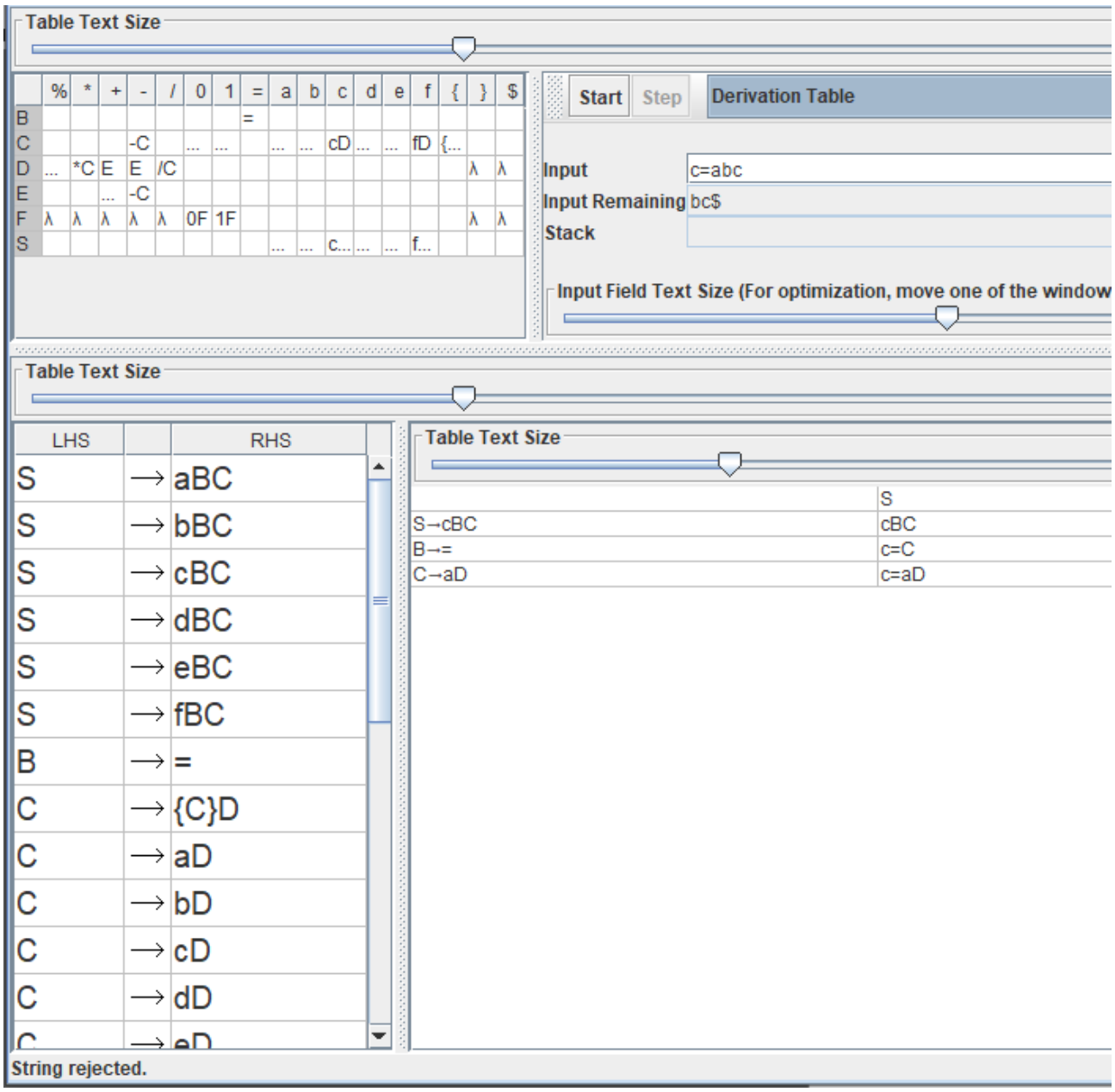

Рисунок 5 – Синтаксический анализ неверной строки

3.2 Программная реализация метода рекурсивного спуска

Была реализована LL(1)-грамматика с помощью системы JFLAP. Она приведена на рисунке 6. На рисунке 7 приведена таблица синтаксического LL(1)-анализа, на рисунке 3 – результат анализа некоторых строк, на рисунках 4 и 5 – подробный процесс синтаксического анализа одной из принимаемых строк и одной из не принимаемых соответственно.

LHS		R
S	→	aBCD;E
S	→	bBCD;E
S	→	cBCD;E
S	→	dBCD;E
S	→	eBCD;E
S	→	fBCD;E
B	→	a
B	→	b
B	→	c
B	→	d
B	→	e
B	→	f
B	→	λ
C	→	=
D	→	{D}F
D	→	aBF
D	→	bBF
D	→	cBF
D	→	dBF
D	→	eBF
D	→	fBF
D	→	'GIF
G	→	0
G	→	1
G	→	2
G	→	3
G	→	4
G	→	5
G	→	6

Рисунок 6 – полученная LL(1)-грамматика, операция унарного минуса обозначена как «@»; идентификаторы систем счисления: «#» - 2 «”» - 8 «’» - 10

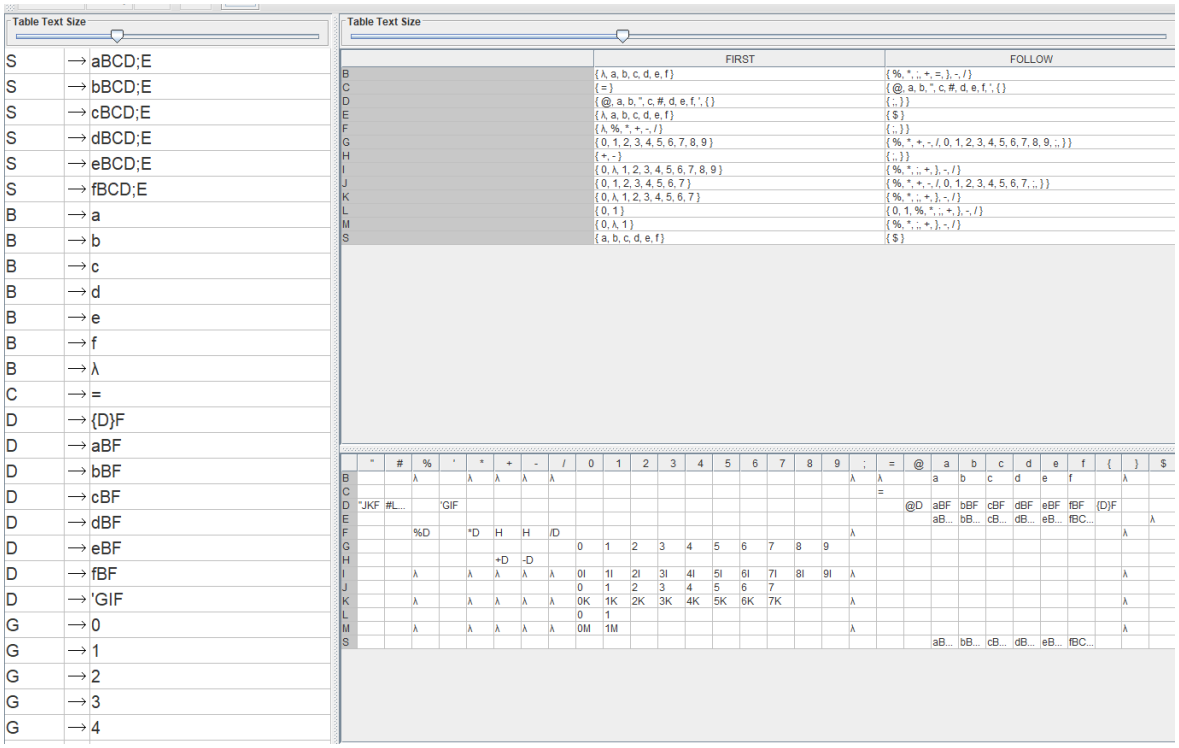

Рисунок 7 – Таблица синтаксического LL(1)-анализа

Для формального доказательства принадлежности грамматики к LL(1)-грамматикам, необходимо показать, что она удовлетворяет следующим условиям.

Для каждого нетерминала A в грамматике генерируется множество терминалов First(A), определенное следующим образом:

- если в грамматике есть правило с A в левой части и правой частью, начинающейся с терминала, то данный терминал входит в First(A);

- если в грамматике есть правило с A в левой части и правой частью, начинающейся с нетерминала (обозначим B), то First(B) строго входит в First(A); никакие иные терминалы не входят в First(A).

Для каждого правила генерируется множество направляющих символов, определенное следующим образом:

- если правая часть правила начинается с терминала, то множество направляющих символов состоит из одного этого терминала;

- иначе правая часть начинается с нетерминала A, тогда множество направляющих символов есть First(A).

Понятно, что First(A) есть объединение множеств направляющих символов для всех правил с A в левой части.

Множество Follow(A) - множество терминальных символов, которые следуют за цепочками, выводимыми из A в грамматике

Грамматика G является LL(1) тогда и только тогда, когда для каждой A-продукции грамматики ($A \to a_1 | \ldots | a_n$, $n > 0$) выполняются следующие условия:

- множества $First(a_1)$, …, $First(a_n)$ попарно не пересекаются (каждая продукция начинается с уникального терминала);

- если $a_i \Rightarrow^* \varepsilon$, то $First(a_j) \cap Follow(A)$ – пустое множество для $1 \leq j \leq n$, $i \neq j$ (множества First и Follow не имеют пересечений для каждой продукции).

Формально грамматика является LL(1)-грамматикой.

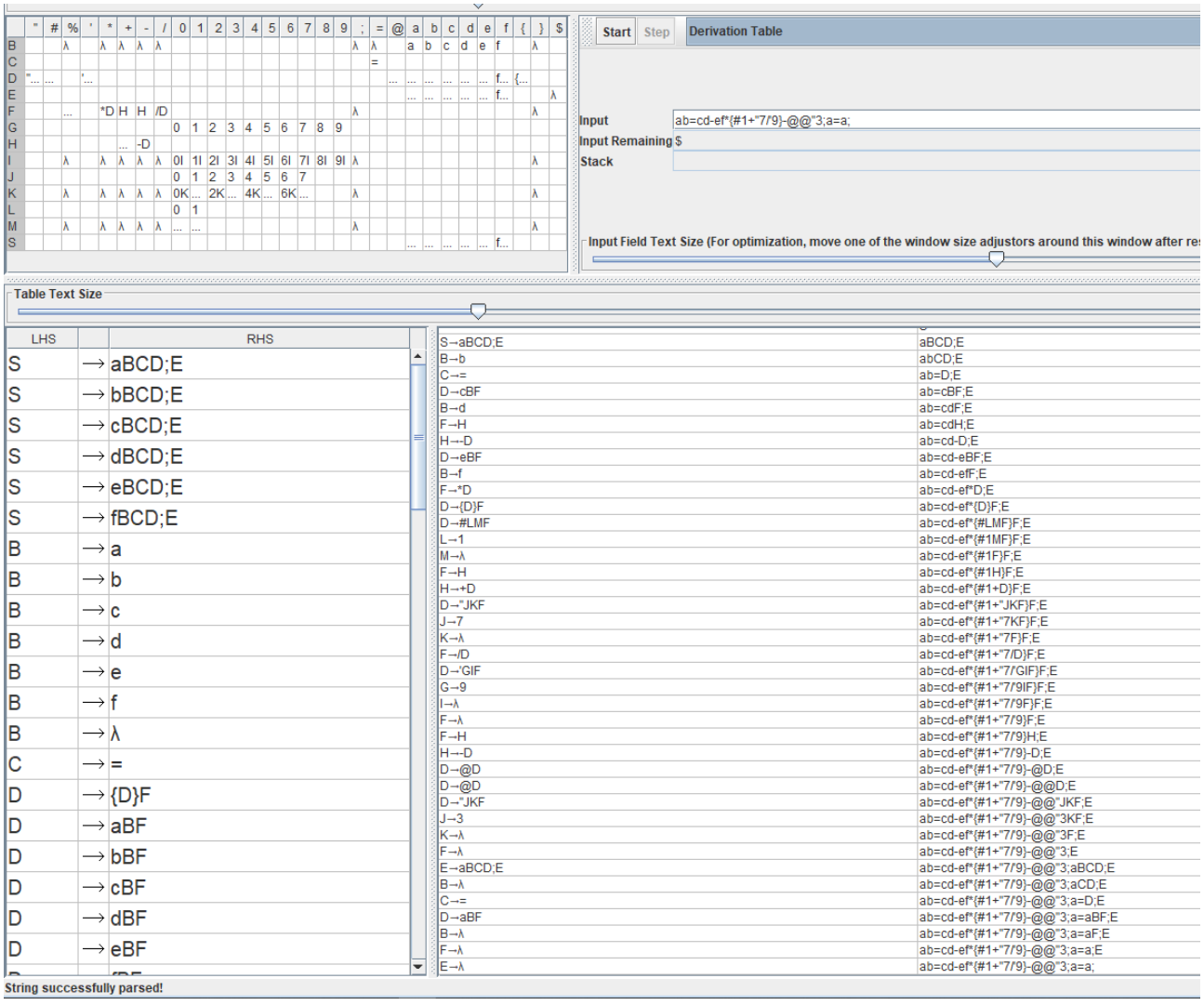

Рисунок 8 – Синтаксический анализ одной из строк

В ходе работы был написан скрипт на языке Python для синтаксического LL(1) анализа строк на основе составленной грамматики. Код скрипта приведен на листинге 1, а примеры работы на рисунках 9-11.

Листинг 1 – код скрипта для синтаксического анализа

```
import sys

EOI = 0
NUM2 = 1.2
BIN_DET = '#'
NUM8 = 1.8
OCT_DET = '"'
NUM10 = 1.10
DEC_DET = '\''
VAR = 2
NEGATIVE = 3
ADDITIVE = 4
MULTI = 5
EQUAL = 6
LB = 20
RB = 21
SEPARATOR = 13
UNKNOWN = -1

TOKEN = 0
LEXEME = 1

SUCCESS = 0
ERROR = -1

MATCHING_DICT = {
    NUM2: ['#', '0', '1'],
    NUM8: ['"', '0', '1', '2', '3', '4',
           '5', '6', '7'],
    NUM10: ['\'', '0', '1', '2', '3', '4',
            '5', '6', '7', '8', '9'],
    VAR: ['a', 'b', 'c', 'd', 'e', 'f'],
    NEGATIVE: ['!'],
    ADDITIVE: ['+', '-'],
    MULTI: ['*', '/', '%'],
    LB: ['('],
    RB: [')'],
    EQUAL: ['='],
    SEPARATOR: [';'],
    EOI: ['$']
}


class RDParser:
    input_index = 0
    str_for_parse = ''
    stack = 'S'

    @staticmethod
    def get_next_token():
        temp_token = EOI

        if RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[NUM2]:
            temp_token = NUM2
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[NUM8]:
            temp_token = NUM8
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[NUM10]:
            temp_token = NUM10
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[VAR]:
            temp_token = VAR
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[LB]:
            temp_token = LB
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[RB]:
            temp_token = RB
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[NEGATIVE]:
            temp_token = NEGATIVE
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[MULTI]:
```

Продолжение листинга 1

```
            temp_token = MULTI
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[ADDITIVE]:
            temp_token = ADDITIVE
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[EQUAL]:
            temp_token = EQUAL
        elif RDParser.str_for_parse[RDParser.input_index] in MATCHING_DICT[SEPARATOR]:
            temp_token = SEPARATOR
        elif len(RDParser.str_for_parse) > RDParser.input_index + 1:
            temp_token = UNKNOWN

        if temp_token == UNKNOWN:
            RDParser.raise_error(Exception('Unknown input symbol!'))

        if temp_token != EOI:
            RDParser.input_index += 1
            return temp_token, RDParser.str_for_parse[RDParser.input_index - 1]
        return temp_token, ''

    @staticmethod
    def token_rollback():
        RDParser.input_index -= 1

    @staticmethod
    def raise_error(exc=Exception('Wrong input!')):
        print('Rejected!')
        raise exc

    @staticmethod
    def parse(str_for_parse):

        RDParser.input_index = 0
        RDParser.stack = 'S'
        RDParser.str_for_parse = str_for_parse + '$'

        if RDParser.start() == 0 and len(str_for_parse) == RDParser.input_index + 1:
            print('Accepted!')
            return True
        else:
            print('Rejected!')
            return False

    @staticmethod
    def print_stack():
        print(RDParser.stack)

    @staticmethod
    def stack_update(func_name, production):
        RDParser.stack = RDParser.stack.replace(func_name, production, 1)

    @staticmethod
    def start():
        res = SUCCESS
        token = RDParser.get_next_token()
        RDParser.print_stack()

        if token[TOKEN] == VAR:
            RDParser.stack_update('S', token[LEXEME] + 'BCD;E')
            RDParser.print_stack()
            res += RDParser.b_func()
            res += RDParser.c_func()
            res += RDParser.d_func()
            if RDParser.get_next_token()[TOKEN] != SEPARATOR:
                RDParser.raise_error()
                # res += ERROR
            res += RDParser.e_func()
        else:
            RDParser.raise_error()
            # res += ERROR
        return res

    @staticmethod
    def b_func():
        token = RDParser.get_next_token()
        if token[TOKEN] == VAR:
            RDParser.stack_update('B', token[LEXEME])
```

Продолжение листинга 1

```
            RDParser.print_stack()
            return SUCCESS
        else:
            RDParser.stack_update('B', '')
            RDParser.print_stack()
            RDParser.token_rollback()
            return SUCCESS

    @staticmethod
    def c_func():
        token = RDParser.get_next_token()
        if token[TOKEN] == EQUAL:
            RDParser.stack_update('C', '=')
            RDParser.print_stack()
            return SUCCESS
        else:
            RDParser.raise_error()
            # return ERROR

    @staticmethod
    def d_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[TOKEN] == LB:
            RDParser.stack_update('D', '{D}F')
            RDParser.print_stack()
            res += RDParser.d_func()
            if RDParser.get_next_token()[TOKEN] != RB:
                RDParser.raise_error()
                # res += ERROR
            res += RDParser.f_func()
        elif token[TOKEN] == VAR:
            RDParser.stack_update('D', token[LEXEME] + 'BF')
            RDParser.print_stack()
            res += RDParser.b_func()
            res += RDParser.f_func()
        elif token[TOKEN] == NUM2 and token[LEXEME] == BIN_DET:
            RDParser.stack_update('D', '#LMF')
            RDParser.print_stack()
            res += RDParser.l_func()
            res += RDParser.m_func()
            res += RDParser.f_func()
        elif token[TOKEN] == NUM8 and token[LEXEME] == OCT_DET:
            RDParser.stack_update('D', '"JKF')
            RDParser.print_stack()
            res += RDParser.j_func()
            res += RDParser.k_func()
            res += RDParser.f_func()
        elif token[TOKEN] == NUM10 and token[LEXEME] == DEC_DET:
            RDParser.stack_update('D', '\'GIF')
            RDParser.print_stack()
            res += RDParser.g_func()
            res += RDParser.i_func()
            res += RDParser.f_func()
        elif token[TOKEN] == NEGATIVE:
            RDParser.stack_update('D', '-D')
            RDParser.print_stack()
            res += RDParser.d_func()
        else:
            RDParser.raise_error()
            # res += ERROR
        return res

    @staticmethod
    def e_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[TOKEN] == VAR:
            RDParser.stack_update('E', token[LEXEME] + 'BCD;E')
            RDParser.print_stack()
            res += RDParser.b_func()
            res += RDParser.c_func()
            res += RDParser.d_func()
            if RDParser.get_next_token()[TOKEN] != SEPARATOR:
                RDParser.raise_error()
```

Продолжение листинга 1

```
                # res += ERROR
            res += RDParser.e_func()
        else:
            RDParser.stack_update('E', '')
            RDParser.print_stack()
            RDParser.token_rollback()
        return res

    @staticmethod
    def f_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[TOKEN] == MULTI:
            RDParser.stack_update('F', token[LEXEME] + 'D')
            RDParser.print_stack()
            res += RDParser.d_func()
        elif token[TOKEN] == ADDITIVE:
            RDParser.stack_update('F', 'H')
            RDParser.print_stack()
            RDParser.token_rollback()
            res += RDParser.h_func()
        else:
            RDParser.stack_update('F', '')
            RDParser.print_stack()
            RDParser.token_rollback()
        return res

    @staticmethod
    def g_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[LEXEME] in MATCHING_DICT[NUM10]:
            RDParser.stack_update('G', token[LEXEME])
            RDParser.print_stack()
        else:
            RDParser.raise_error()
        return res

    @staticmethod
    def h_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[TOKEN] == ADDITIVE:
            RDParser.stack_update('H', token[LEXEME] + 'D')
            RDParser.print_stack()
            res += RDParser.d_func()
        else:
            RDParser.raise_error()
        return res

    @staticmethod
    def i_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[LEXEME] in MATCHING_DICT[NUM10]:
            RDParser.stack_update('I', token[LEXEME] + 'I')
            RDParser.print_stack()
            res += RDParser.i_func()
        else:
            RDParser.stack_update('I', '')
            RDParser.print_stack()
            RDParser.token_rollback()
        return res

    @staticmethod
    def j_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[LEXEME] in MATCHING_DICT[NUM8]:
            RDParser.stack_update('J', token[LEXEME])
            RDParser.print_stack()
        else:
            RDParser.raise_error()
        return res
```

Окончание листинга 1

```
    @staticmethod
    def k_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[LEXEME] in MATCHING_DICT[NUM8]:
            RDParser.stack_update('K', token[LEXEME] + 'K')
            RDParser.print_stack()
            res += RDParser.k_func()
        else:
            RDParser.stack_update('K', '')
            RDParser.print_stack()
            RDParser.token_rollback()
        return res

    @staticmethod
    def l_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[LEXEME] in MATCHING_DICT[NUM2]:
            RDParser.stack_update('L', token[LEXEME])
            RDParser.print_stack()
        else:
            RDParser.raise_error()
        return res

    @staticmethod
    def m_func():
        res = SUCCESS
        token = RDParser.get_next_token()
        if token[LEXEME] in MATCHING_DICT[NUM2]:
            RDParser.stack_update('M', token[LEXEME] + 'M')
            RDParser.print_stack()
            res += RDParser.m_func()
        else:
            RDParser.stack_update('M', '')
            RDParser.print_stack()
            RDParser.token_rollback()
        return res


def main():
    if len(sys.argv) > 1:
        try:
            RDParser.parse(sys.argv[1])
        except Exception as e:
            print(e)
    else:
        temp = input()
        try:
            RDParser.parse(temp)
        except Exception as e:
            print(e)


if __name__ == "__main__":
    main()
```

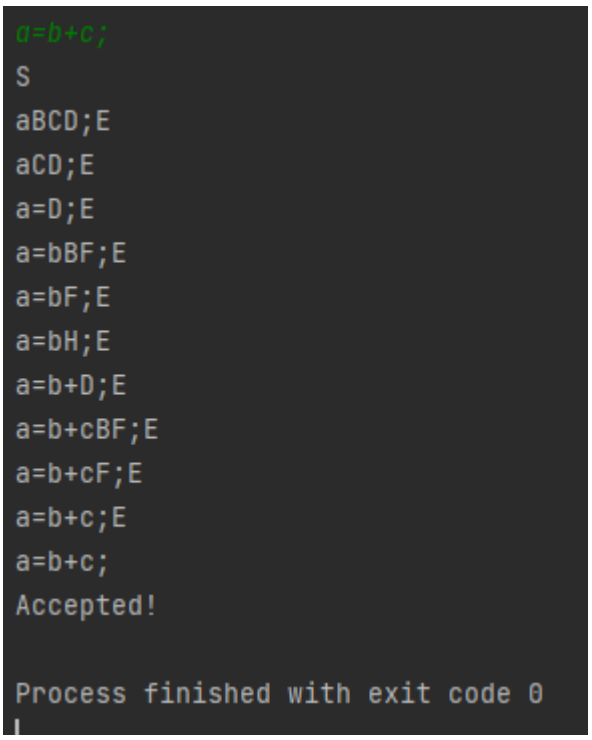

Рисунок 9 – Пример работы скрипта (подходящая строка)

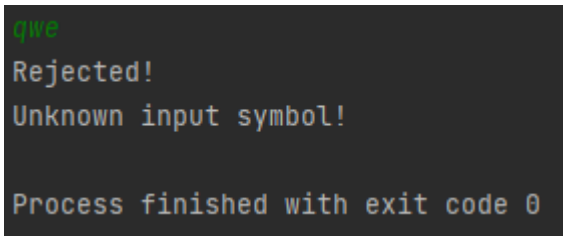

Рисунок 10 – Пример работы скрипта (символ не из алфавита)

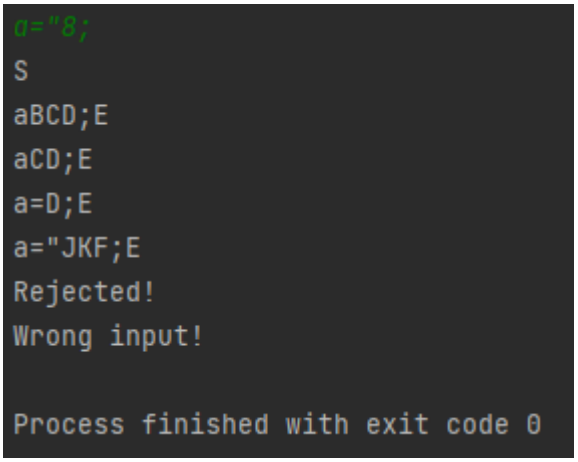

Рисунок 11 – Пример работы скрипта (неподходящая строка)

3.3 SLR-грамматика

Была построена SLR-грамматика в соответствии с заданием. Она приведена на рисунказ 12-14.

LHS		RHS
S	→	L=R;Z
Z	→	L=R;Z
Z	→	λ
L	→	a
L	→	b
L	→	c
L	→	d
L	→	e
L	→	f
R	→	-R
R	→	{R}A
A	→	*R
A	→	/R
A	→	%R
A	→	B
B	→	+R
B	→	-R
B	→	λ
R	→	CA
C	→	a
C	→	b
C	→	c
C	→	d
C	→	e
C	→	f
C	→	'DE
C	→	#FG
D	→	0
D	→	1
D	→	2
D	→	3

Рисунок 12 – Полученная грамматика для SLR анализа (идентификаторы систем счисления: «#» - 2, «’» - 10)

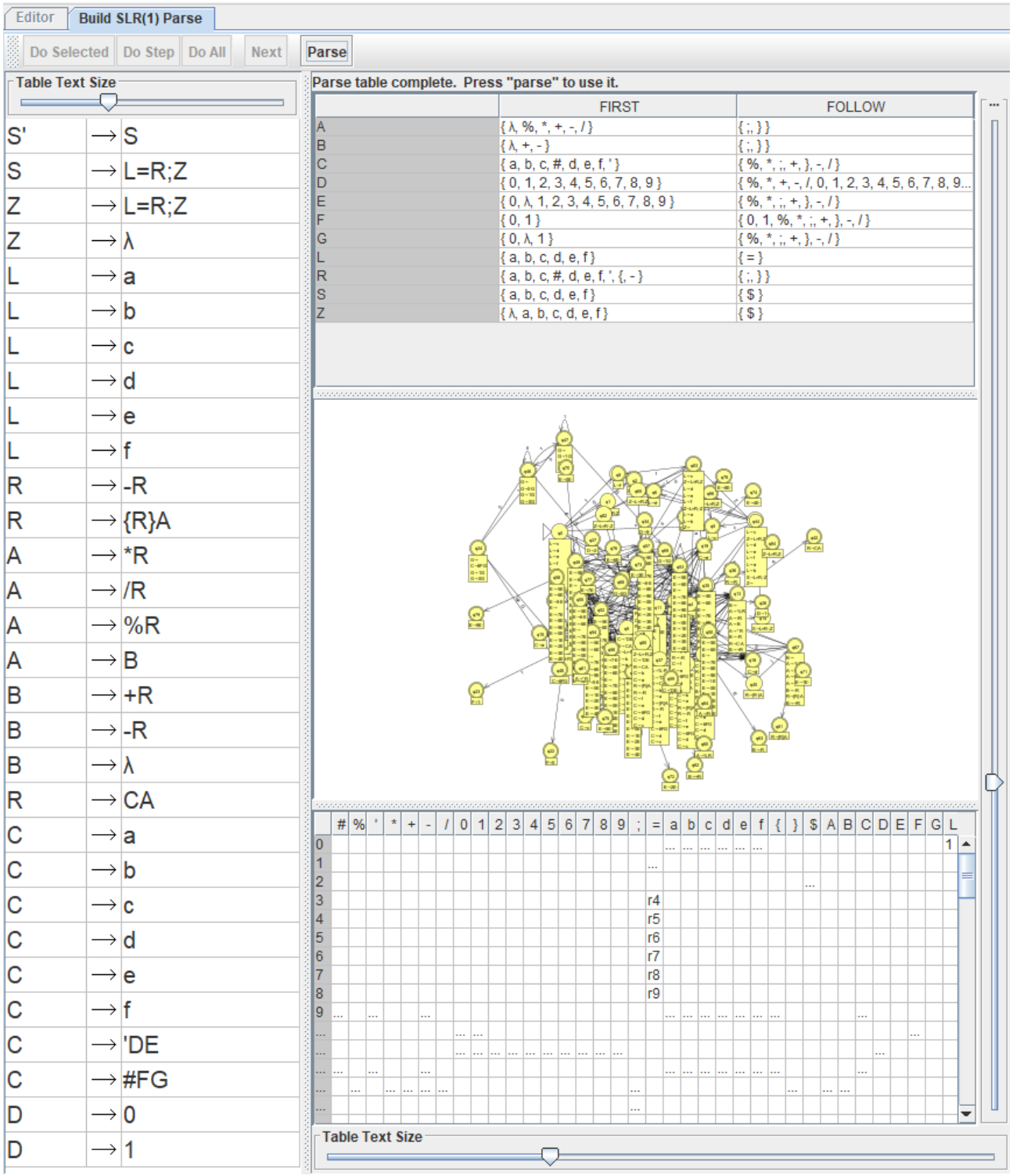

Рисунок 13 – Множества FIRST и FOLOW, TCA (PDA.jff)

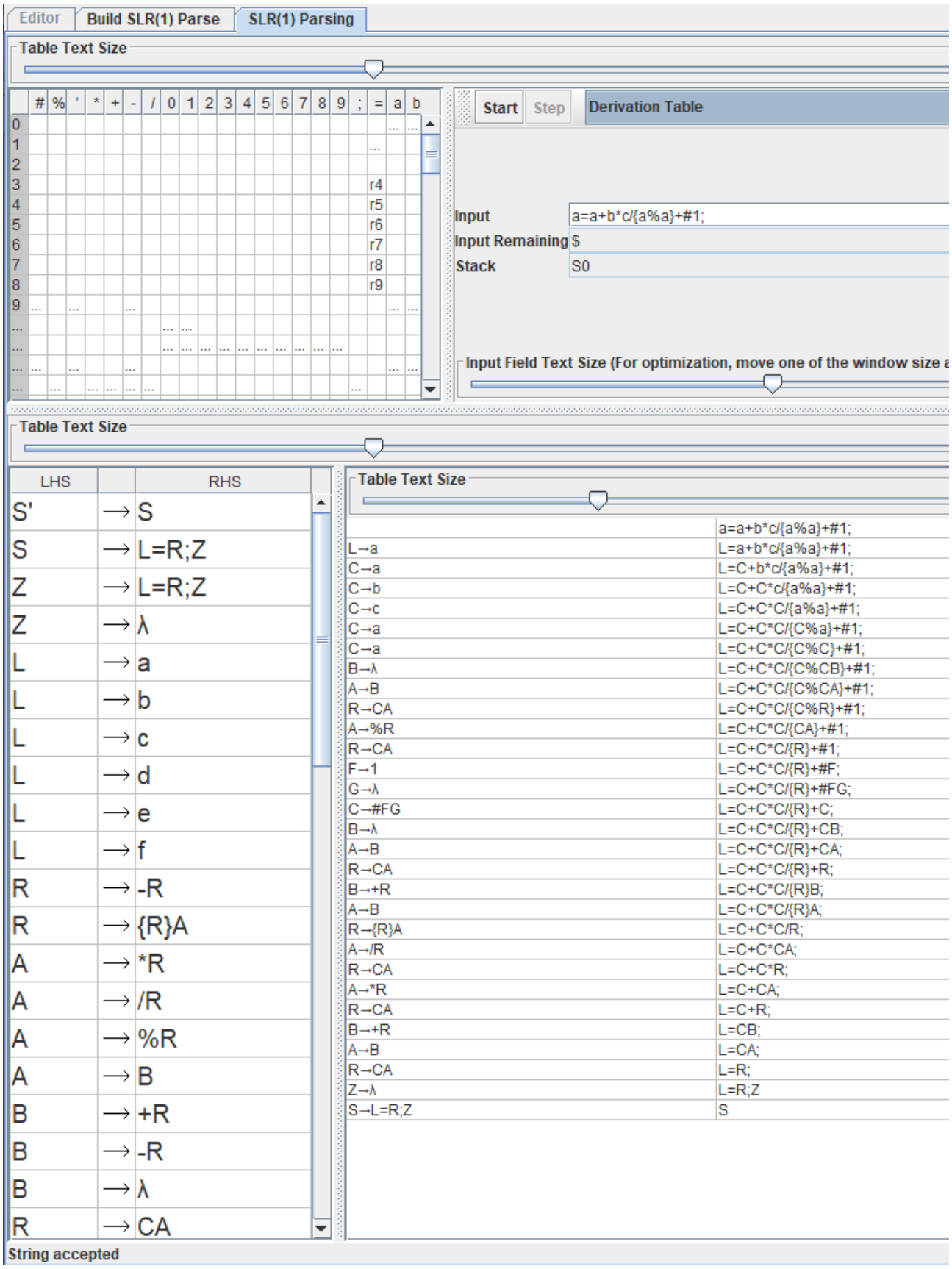

Рисунок 14 – Синтаксический анализ одной из строк

4 Вывод

В ходе данной лабораторной работы были исследованы свойства универсальных алгоритмов синтаксического анализа контекстно-свободных языков.