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

Институт Космических и информационных технологий
институт
Кафедра «Информатика»
кафедра

ОТЧЕТ О ПРАКТИЧЕСКОЙ РАБОТЕ №5

Синтаксический анализ контекстно-свободных языков
тема

Преподаватель _____ А. С. Кузнецов
подпись, дата инициалы, фамилия

Студент КИ18-17/1б 031830504 _____ Е.В. Железкин
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 Задача работы

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.

Вариант (1, 1, 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.

2.1 Инструкция по запуску

Необходимо установить *python*, желательно версии 3 и выше (выполнено на версии 3.9.4):

- Страница загрузки для Windows: https://www.python.org/downloads/
- Для Linux есть несколько способов, один из них инструмент apt-get:

  $ sudo apt-get update

  $ sudo apt-get install python3.8
- Или загрузить, распаковать и установить образ:

$ wget https://www.python.org/ftp/python/3.8.2/Python-3.8.2.tgz

$ tar -xvf Python-3.8.2.tgz

Для следующего шага понадобится компилятор gcc, но, думаю, это не проблема. Переходим в распакованную папку и собираем+устанавливаем:

$ cd Python-3.8.2

$ ./configure

$ make

$ sudo make install

Далее на любой из двух систем перейти в каталог с распакованным архивом Lab_5 и выполнить:

$ python recursive_descent_parser.py <argv>

$ pytest RDP_tests.py

Реализация поддерживает как ввод тестовой строки как в качестве аргумента командной строки, так и непосредственно во входном потоке.

Для запуска тестов:

Установить библиотеку pytest:

$ pip install pytest

Запуск:

$ pytest RDP_tests.py

3 Ход работы

Часть 1

Реализована LL(1)-грамматика с помощью системы JFLAP:

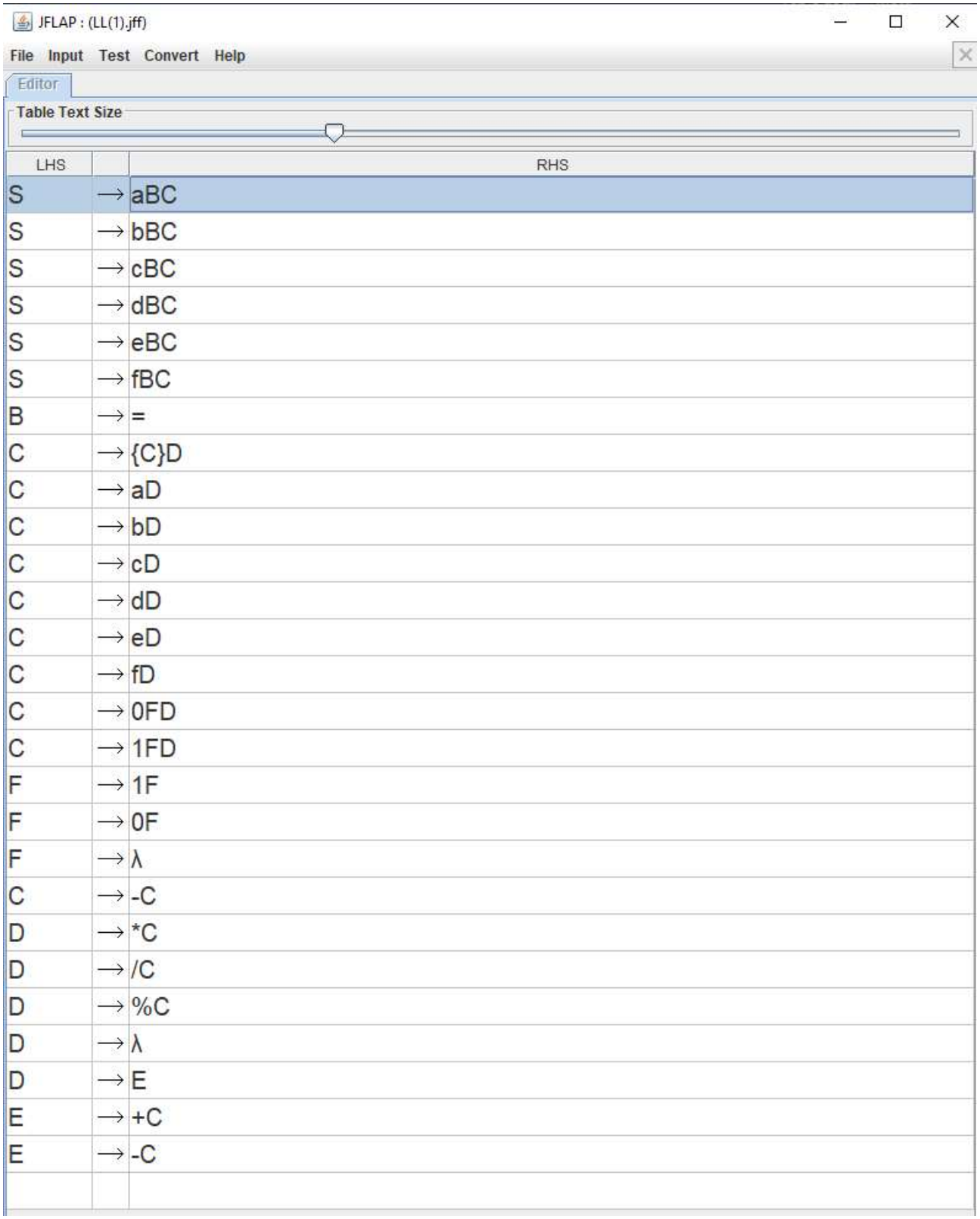


Рисунок 1 – полученная LL(1)-грамматика (файл LL(1)-1.jff)

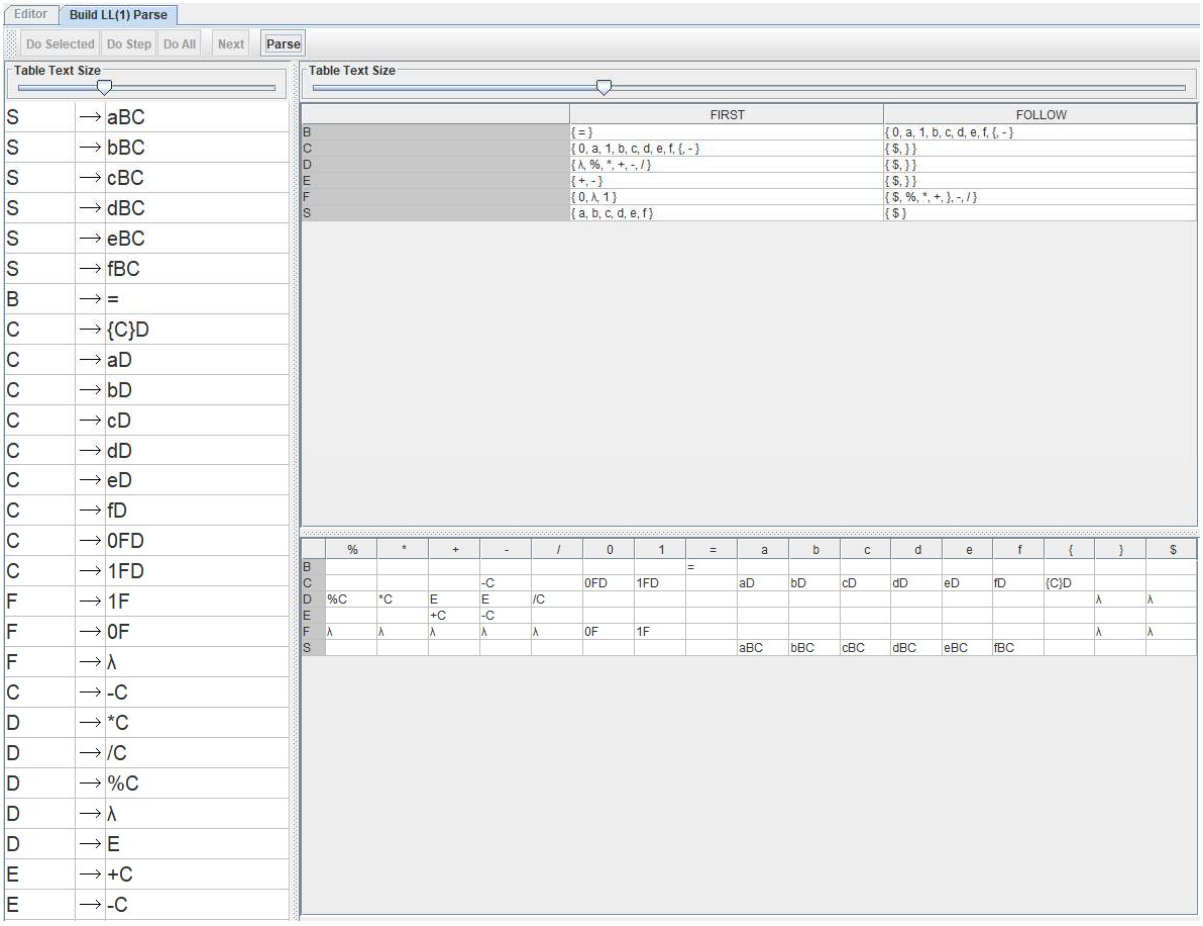


Рисунок 2 – таблица синтаксического LL(1)-анализа

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 – Тестирование полученной грамматики

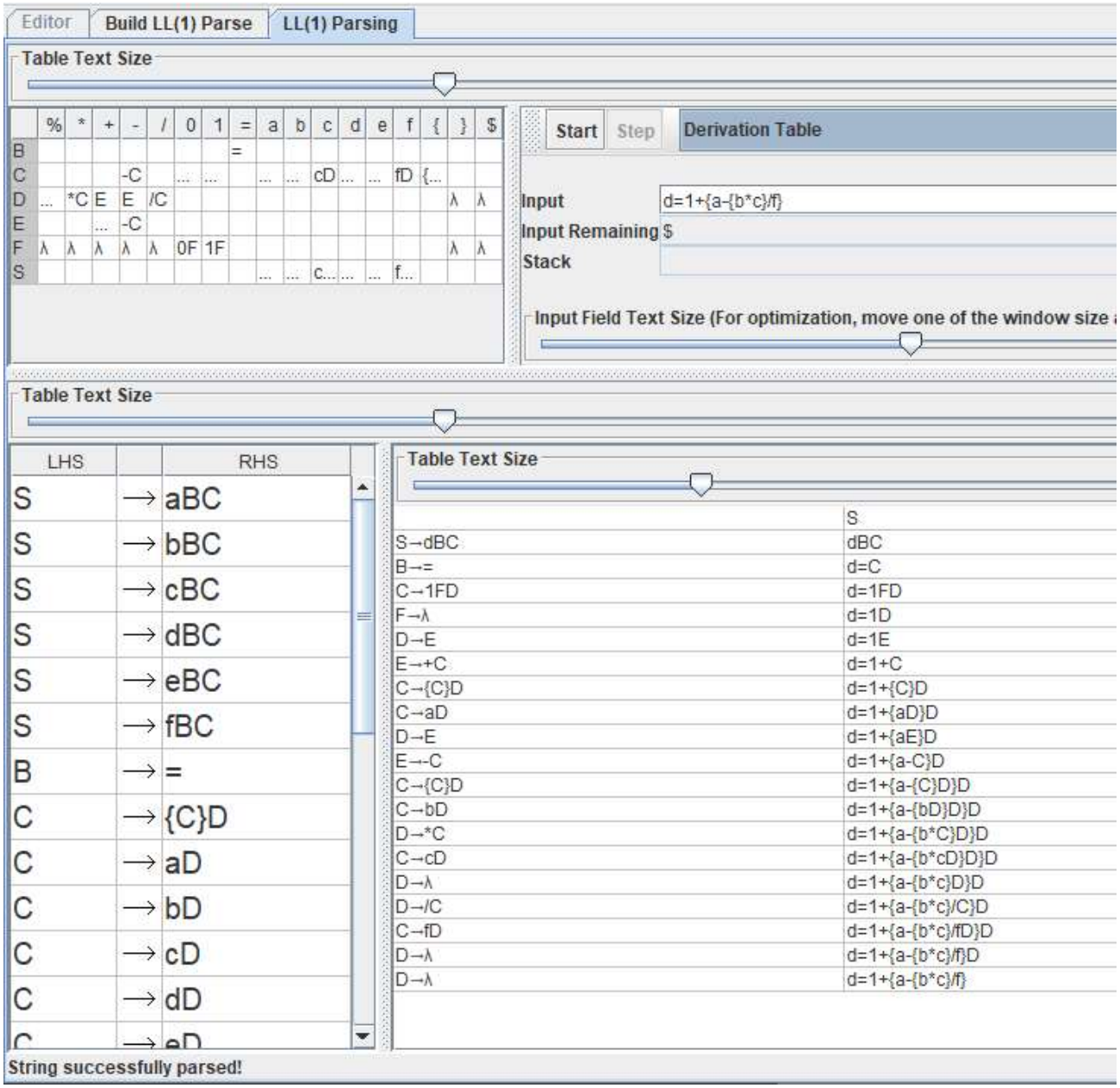


Рисунок 4 – Перехват экрана распознавания

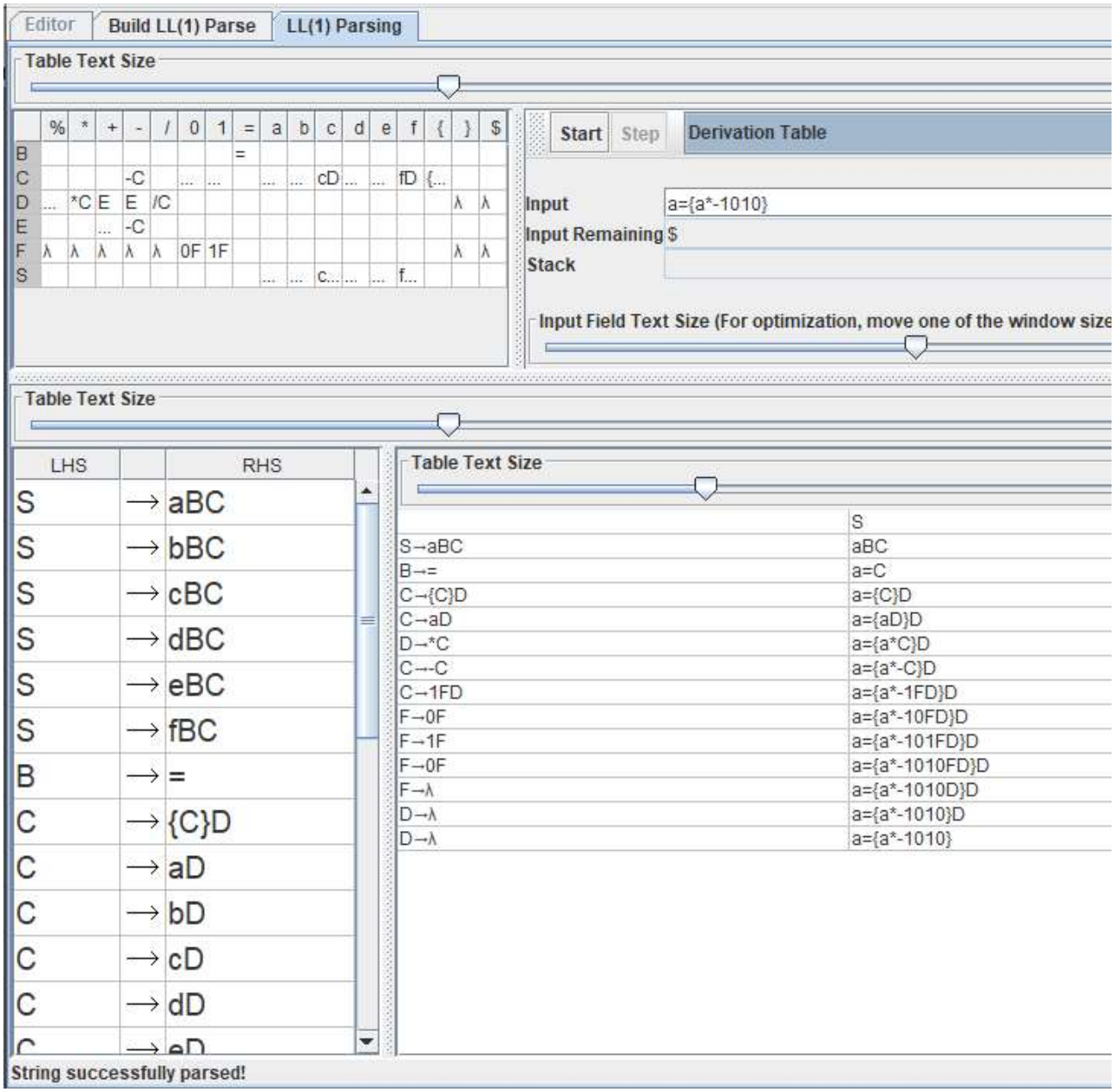


Рисунок 5 – Перехват экрана распознавания

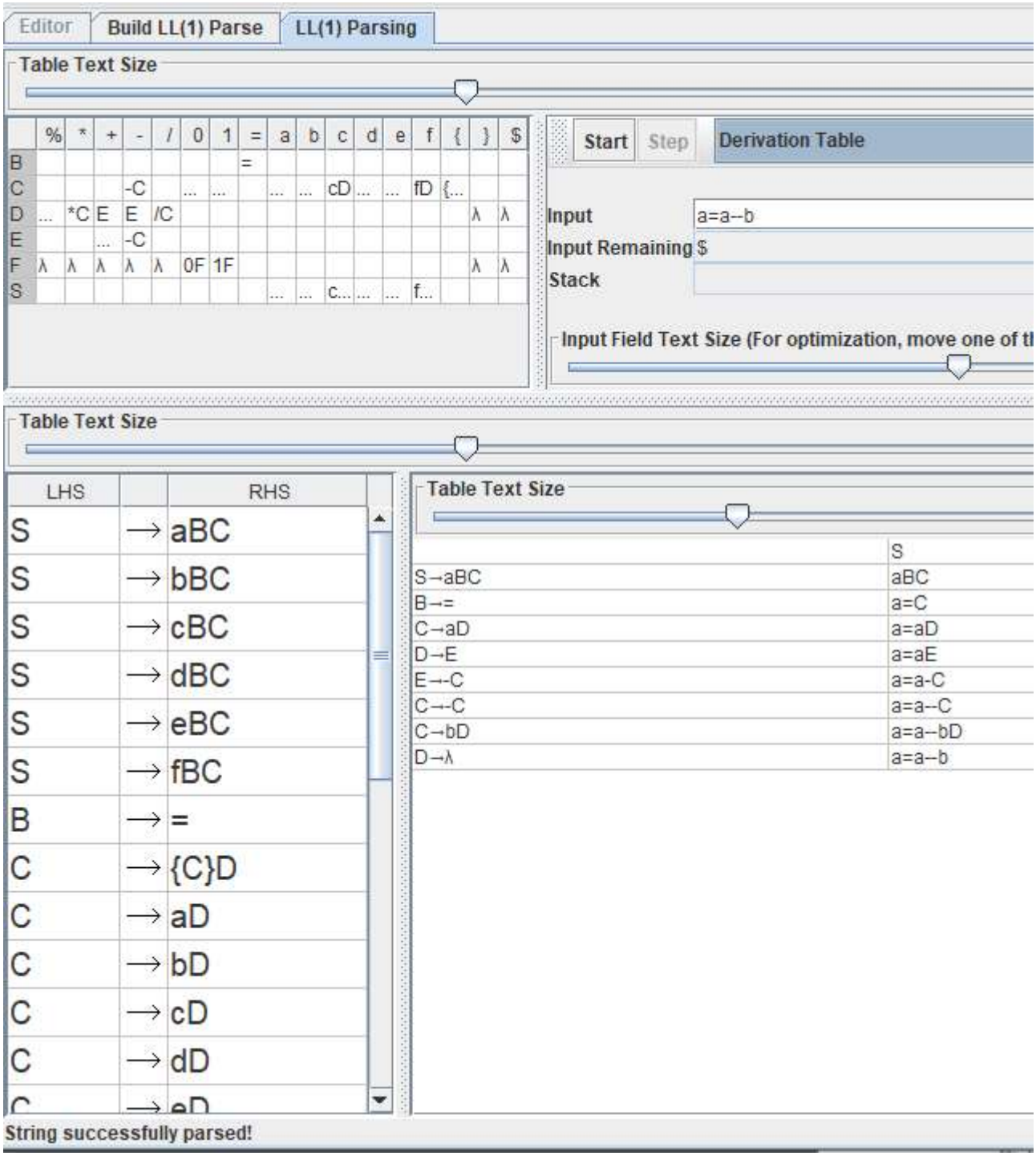


Рисунок 6 – Перехват экрана распознавания

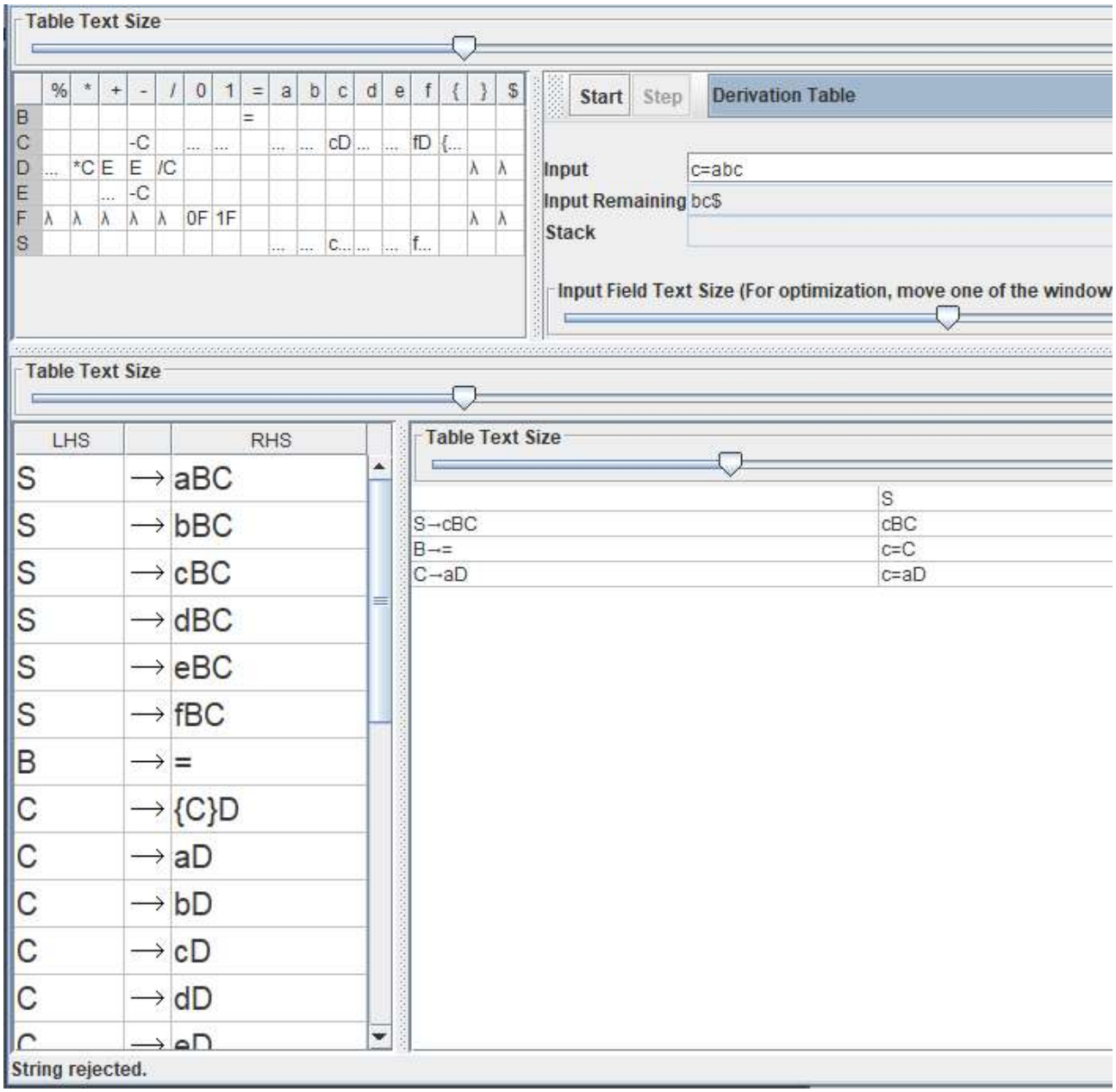
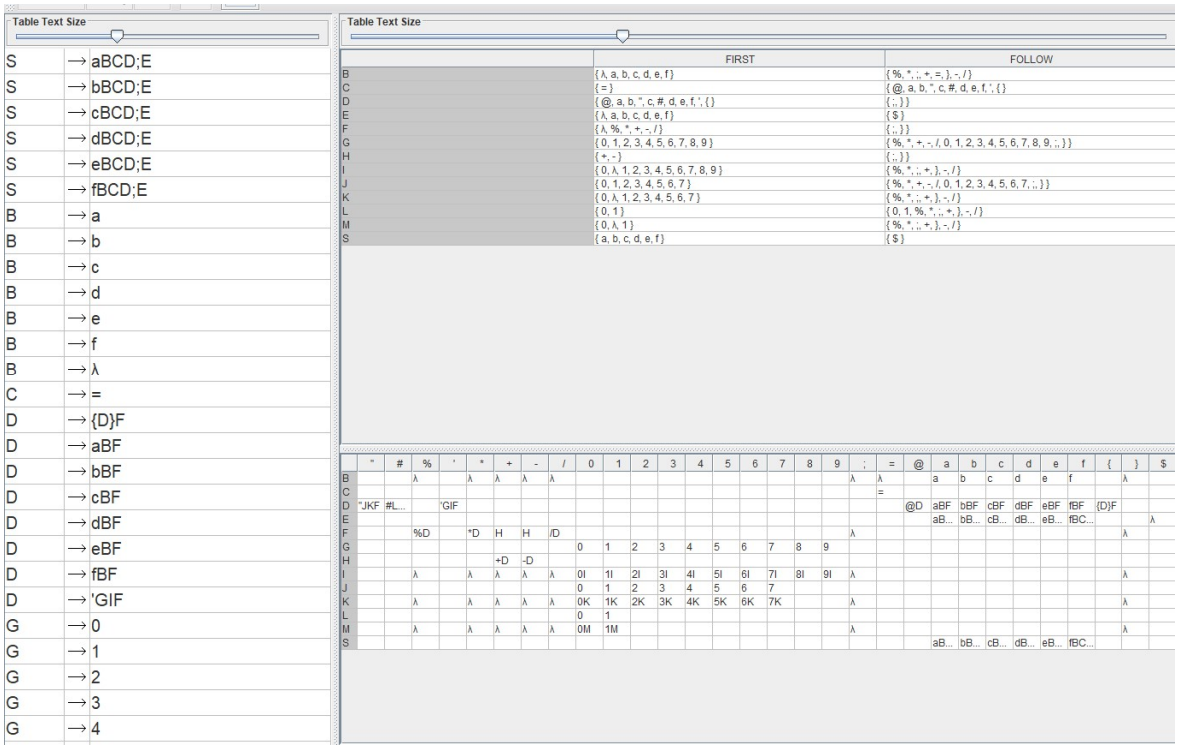


Рисунок 7 – Перехват экрана распознавания (для неверной строки)

Часть 2

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



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

	FIRST	FOLLOW
B	{ λ, a, b, c, d, e, f }	{ %, *, ;, +, =,), -, / }
C	{ = }	{ @, a, b, ", c, #, d, e, f, ', { }
D	{ @, a, b, ", c, #, d, e, f, ', { }	{ ;, } }
E	{ λ, a, b, c, d, e, f }	{ $ }
F	{ λ, %, *, +, -, / }	{ ;, } }
G	{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }	{ %, *, +, -, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ;, } }
H	{ +, - }	{ ;, } }
I	{ 0, λ, 1, 2, 3, 4, 5, 6, 7, 8, 9 }	{ %, *, ;, +,), -, / }
J	{ 0, 1, 2, 3, 4, 5, 6, 7 }	{ %, *, +, -, /, 0, 1, 2, 3, 4, 5, 6, 7, ;, } }
K	{ 0, λ, 1, 2, 3, 4, 5, 6, 7 }	{ %, *, ;, +,), -, / }
L	{ 0, 1 }	{ 0, 1, %, *, ;, +,), -, / }
M	{ 0, λ, 1 }	{ %, *, ;, +,), -, / }
S	{ a, b, c, d, e, f }	{ $ }

	"	#	%	'	*	+	-	/	0	1	2	3	4	5	6	7	8	9	;	=	@	a	b	c	d	e	f	{	}	$
B			λ		λ	λ	λ	λ											λ	λ		a	b	c	d	e	f		λ	
C																				=										
D	"JKF	#L...		'GIF																	@D	aBF	bBF	cBF	dBF	eBF	fBF	{D}F		
E																						aB...	bB...	cB...	dB...	eB...	fBC...			λ
F			%D		*D	H	H	/D											λ										λ	
G									0	1	2	3	4	5	6	7	8	9											λ	
H						+D	-D																							
I			λ		λ	λ	λ	λ	0I	1I	2I	3I	4I	5I	6I	7I	8I	9I	λ										λ	
J									0	1	2	3	4	5	6	7														
K			λ		λ	λ	λ	λ	0K	1K	2K	3K	4K	5K	6K	7K			λ										λ	
L									0	1																				
M			λ		λ	λ	λ	λ	0M	1M									λ										λ	
S																						aB...	bB...	cB...	dB...	eB...	fBC...			

Рисунок 9 – Таблица CA

Грамматика, удовлетворяющая следующим правилам, считается LL(1):

- G - LL(1) тогда и только тогда, когда для каждой A-продукции грамматики $(A \to a_1 \mid \ldots \mid a_n,\ n > 0)$ выполняются следующие условия:
  - Множества $FIRST(a_1), \ldots, FIRST(a_n)$ попарно не пересекаются
  - Если $a_i \Rightarrow^* \varepsilon$, то $FIRST(a_j) \cap FOLLOW(A) = \varnothing$ для $1 \le j \le n,\ i \ne j$

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)

Возможны обобщения этих определений для случая наличия правил вида `A → null`.

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

Грамматика *разбираема по LL(1)*, если для любой пары правил с одинаковой левой частью множества направляющих символов не пересекаются.

Рисунок 10 – Правила, которым удовлетворяет LL(1)-грамматика

Для того, чтобы формально доказать, что описанная грамматика является LL(1)-грамматикой, обратимся к правилам, описанным выше:

- множества FIRST(a1),.., FIRST(an) попарно не пересекаются(каждая продукция начинается с уникального терминала);
- множества FIRST и FOLLOW не имеют пересечений для каждой продукции)

Формально грамматика является LL(1)-грамматикой.

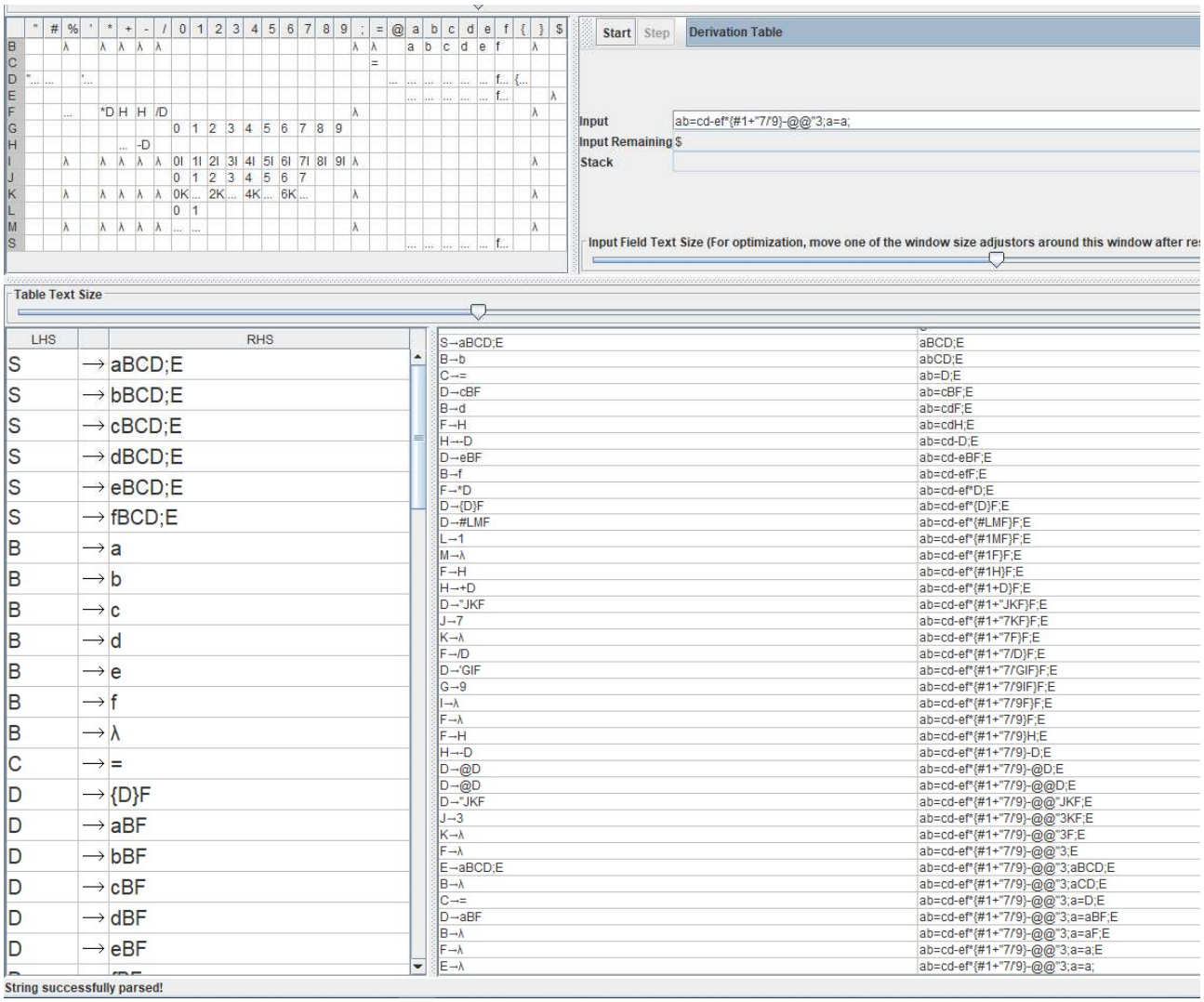


Рисунок 11 – Перехват экрана распознавания

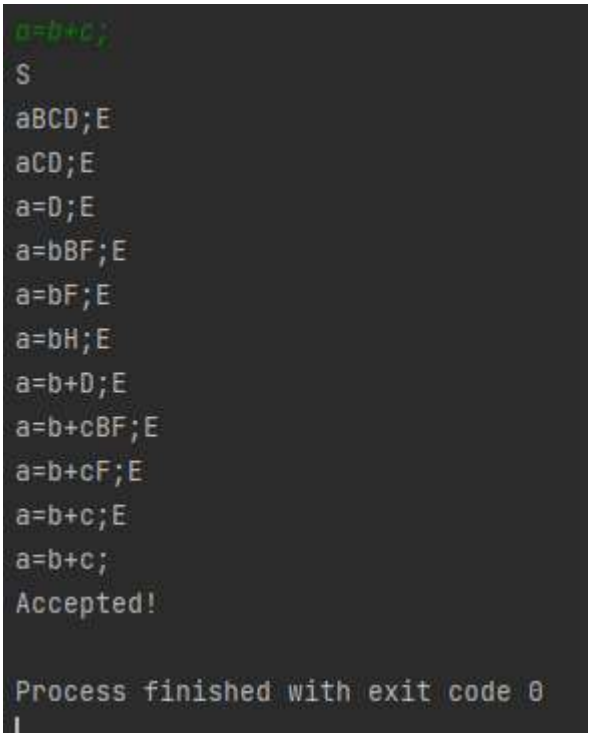


Рисунок 12 – Наглядное тестирование программной реализации

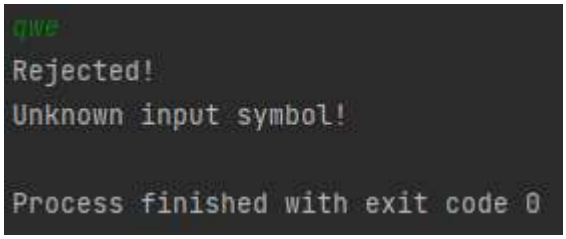


Рисунок 13 – Наглядное тестирование программной реализации

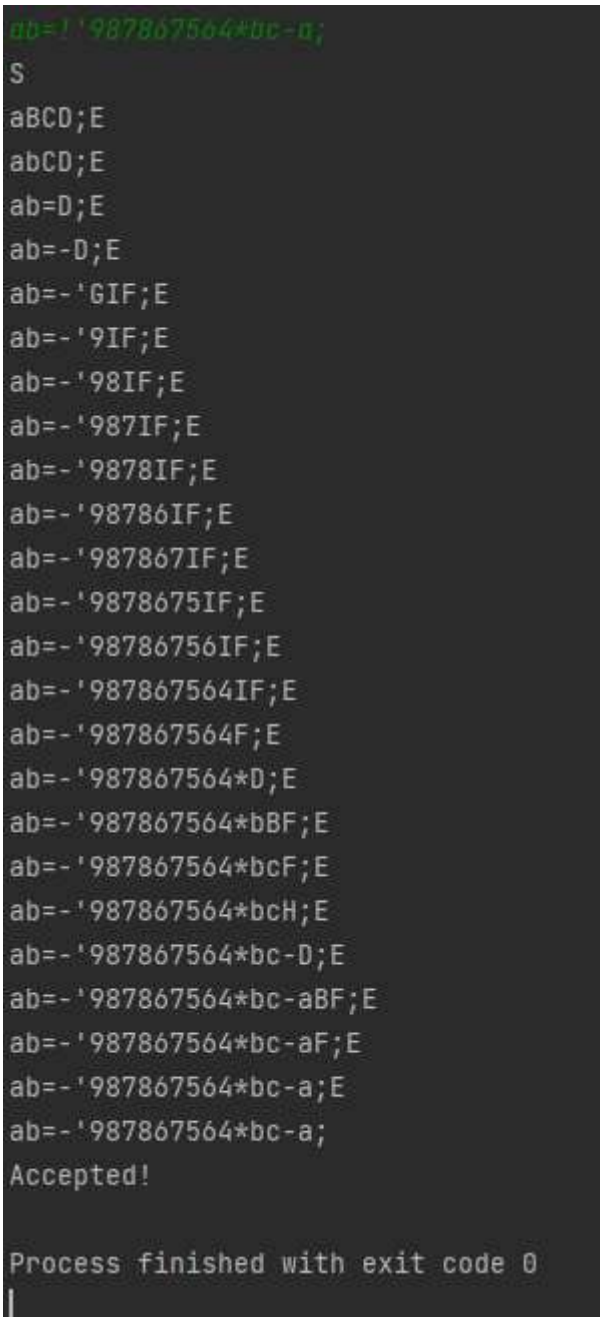


Рисунок 14 – Наглядное тестирование программной реализации

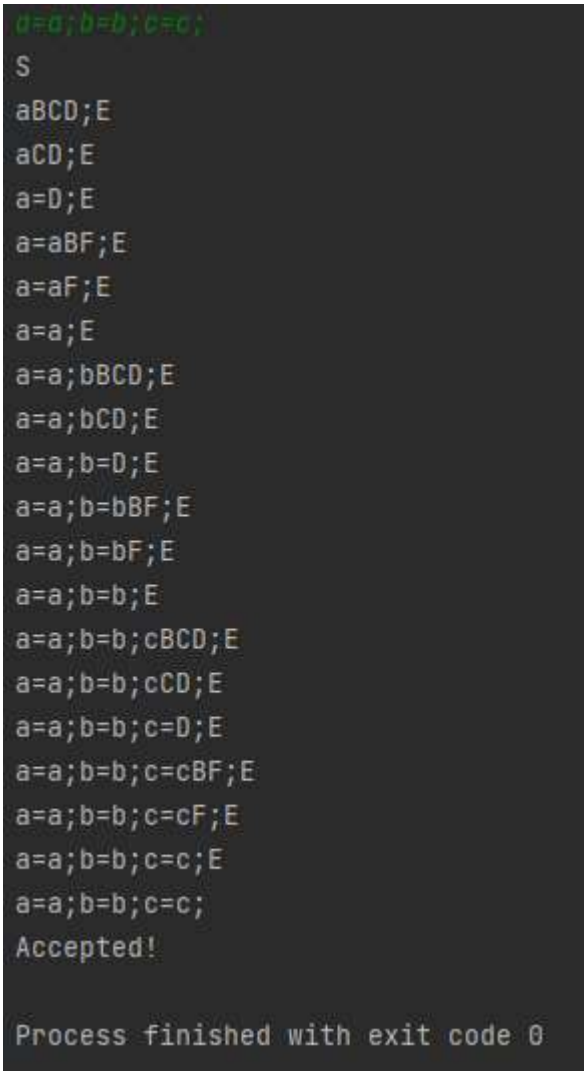


Рисунок 15 – Наглядное тестирование программной реализации

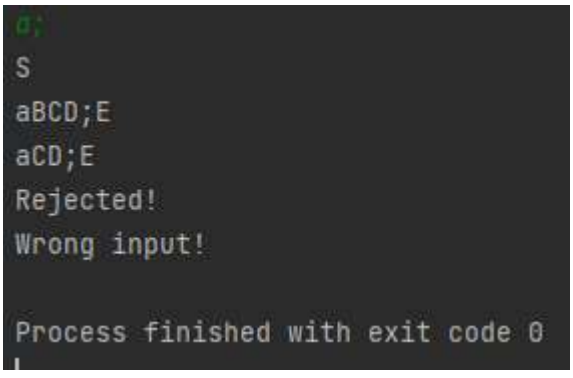


Рисунок 16 – Наглядное тестирование программной реализации

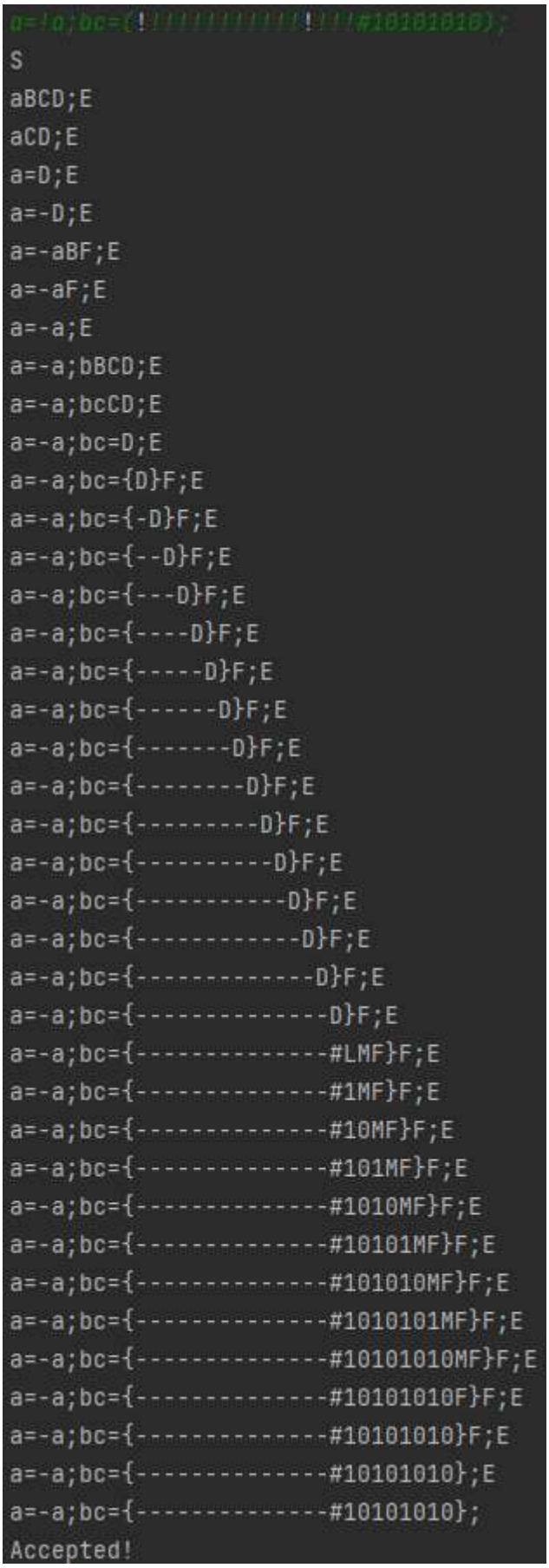


Рисунок 17 – Наглядное тестирование программной реализации

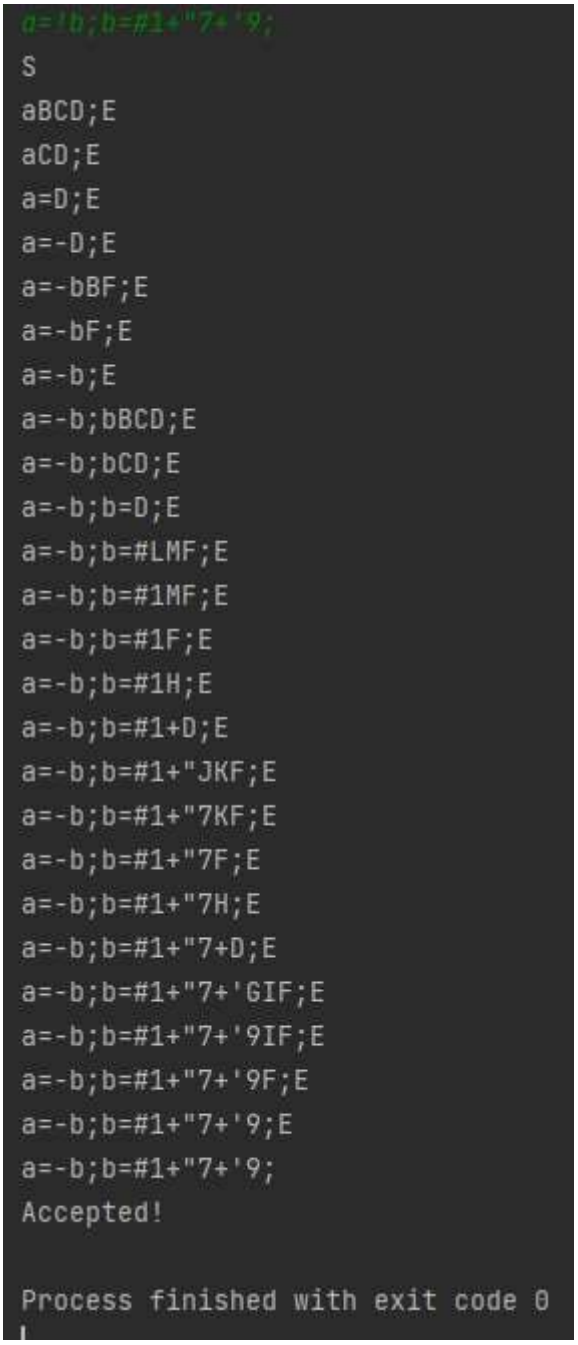


Рисунок 18 – Наглядное тестирование программной реализации

Рисунок 19 – Наглядное тестирование программной реализации

Рисунок 20 – Наглядное тестирование программной реализации

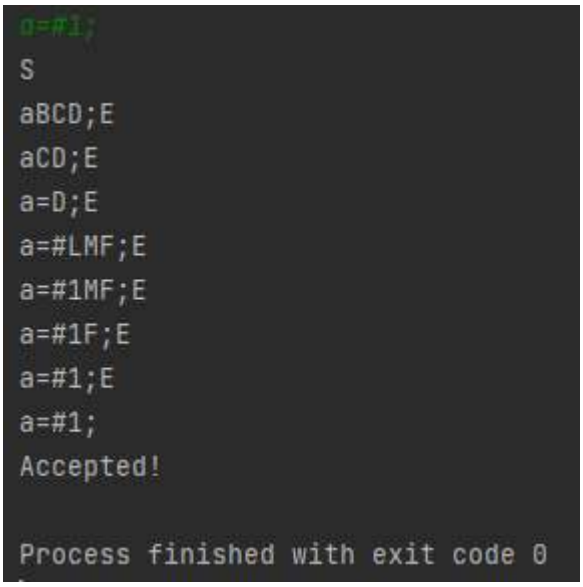


Рисунок 21 – Наглядное тестирование программной реализации

Рисунок 22 – Наглядное тестирование программной реализации

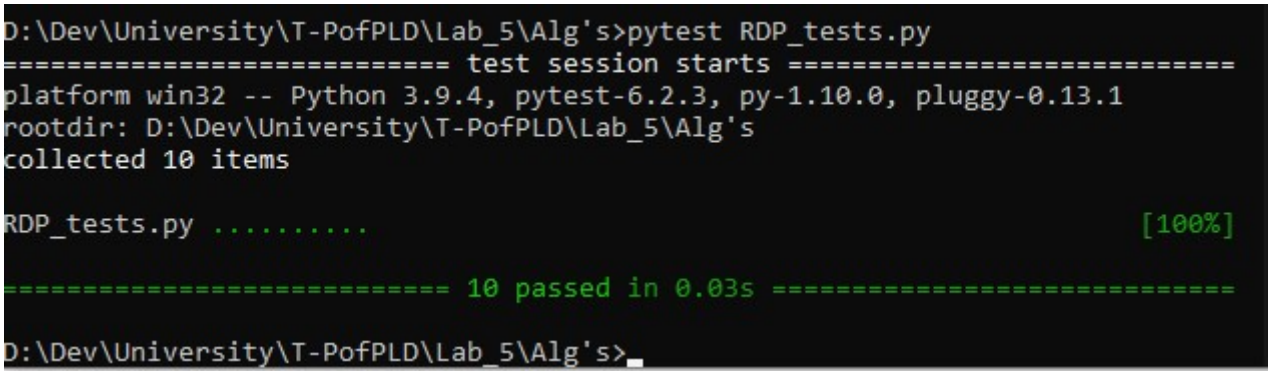


Рисунок 23 – Тестирование программной реализации с помощью набора тестов

Для запуска тестов необходимо установить библиотеку *pytest* (pip install pytest) для python и выполнить *pytest CYK_tests.py*

Часть 3

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 24 – Полученная грамматика для SLR анализа (SLR.jff; идентификаторы систем счисления: «#» - 2, «’» - 10)

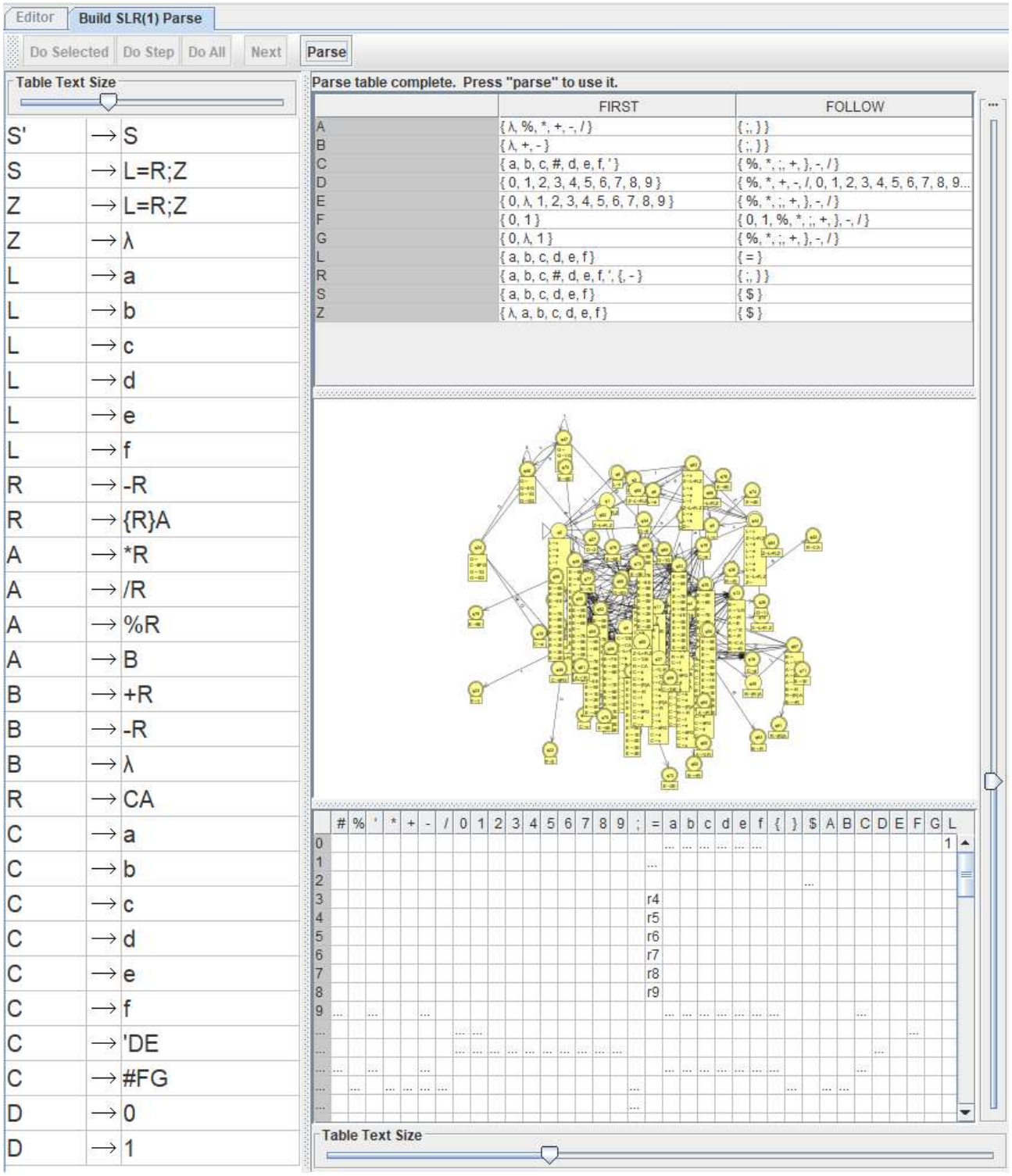


Рисунок 25 – Множества FIRST и FOLOW, ТСА (PDA.jff)

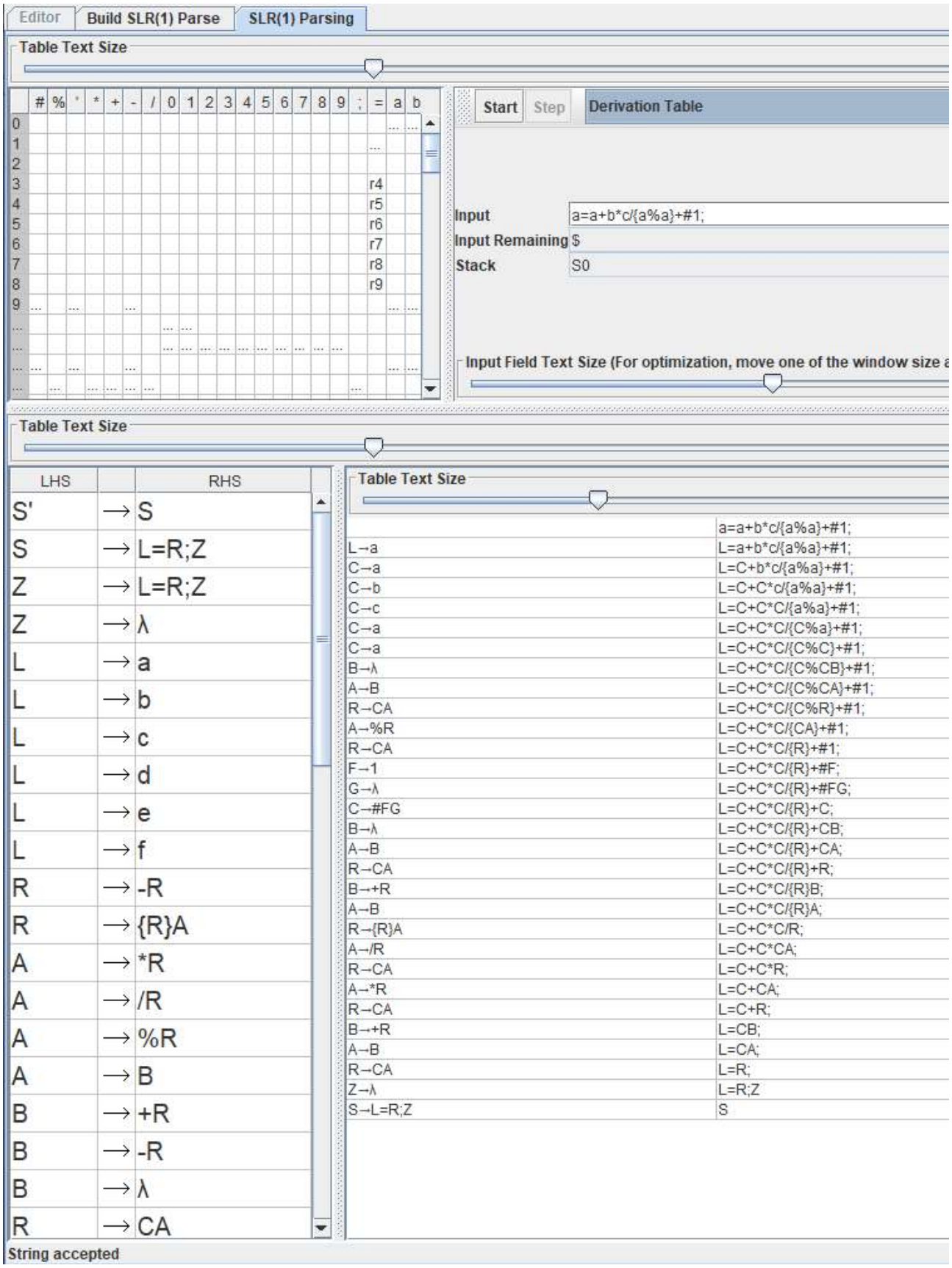


Рисунок 26 – Перехват экрана распознавания

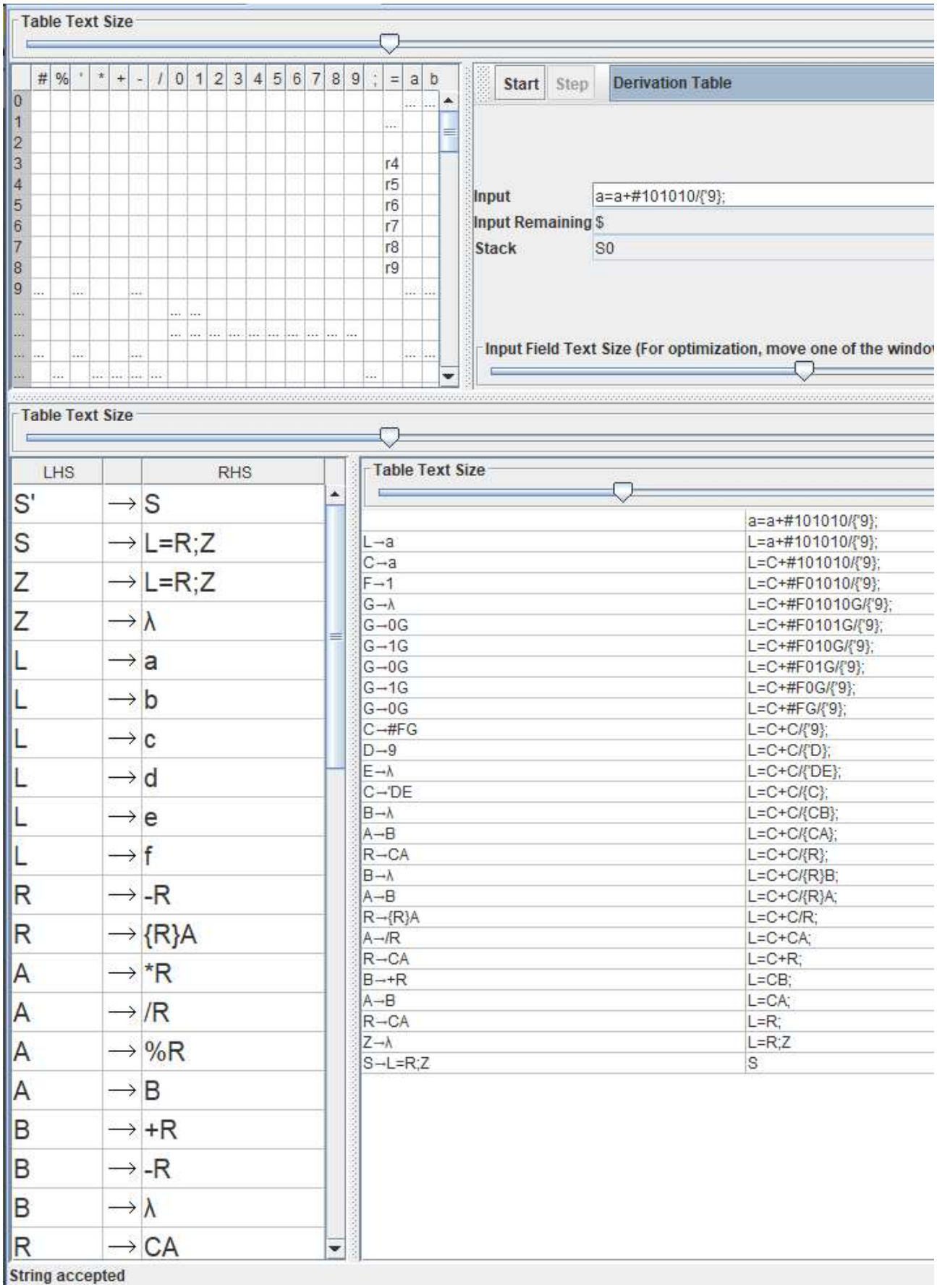


Рисунок 27 – Перехват экрана распознавания

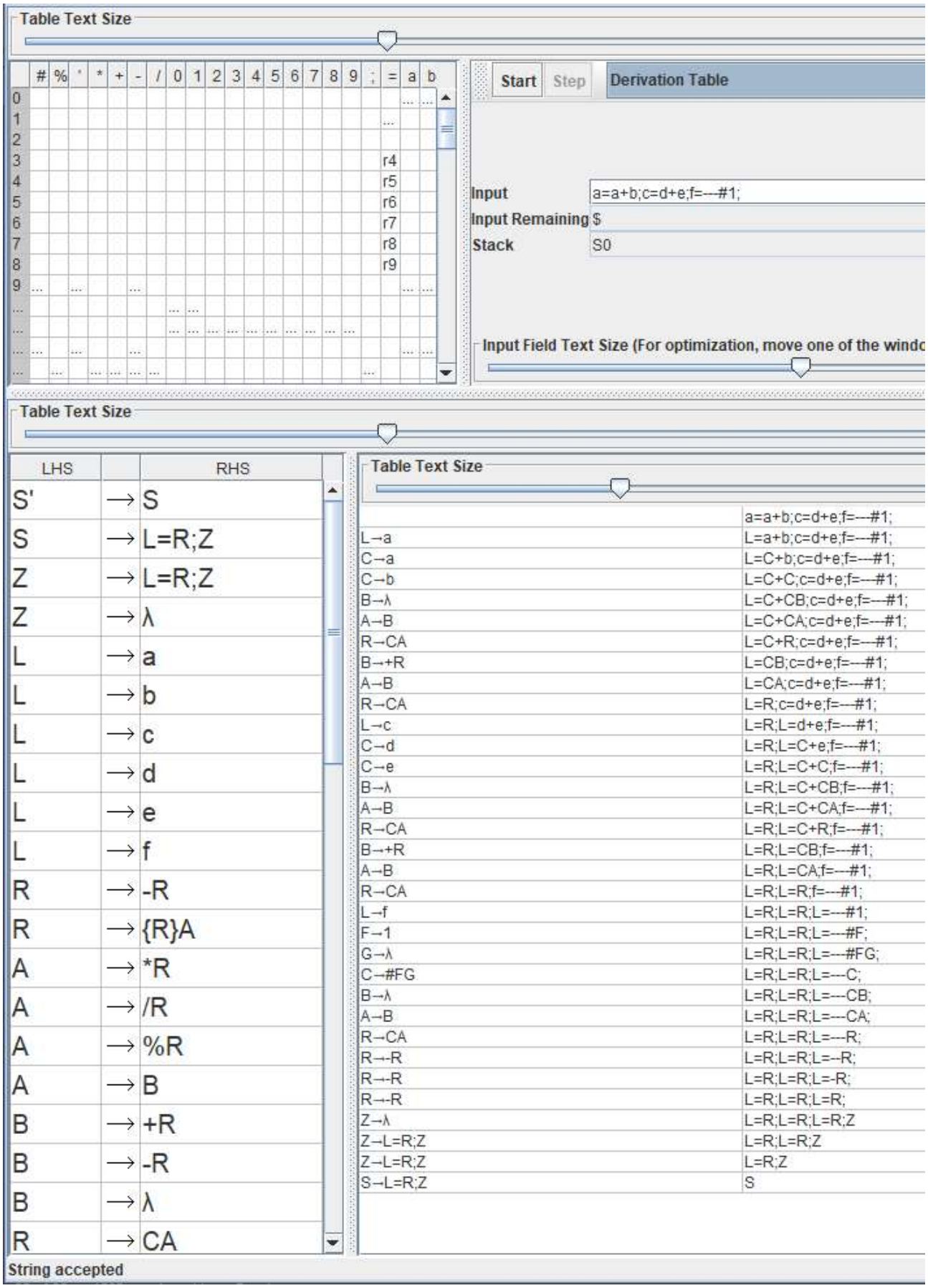


Рисунок 28 – Перехват экрана распознавания

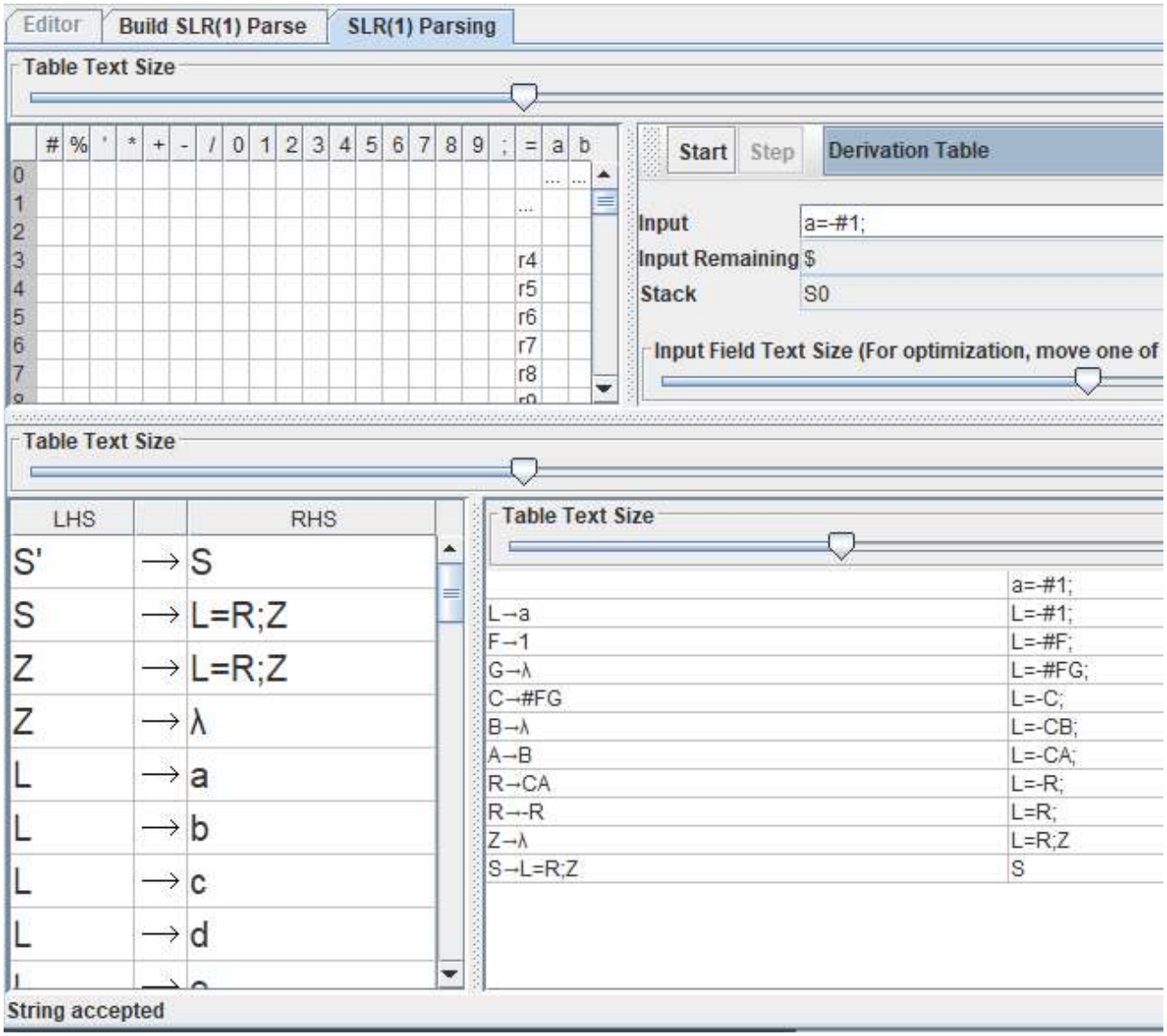


Рисунок 29 – Перехват экрана распознавания

4 Вывод

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

ПРИЛОЖЕНИЕ А

Листинг 1 – файл recursive_descent_parser.py

```
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]:
                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])
        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()
            # 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

    @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()
```

Листинг 2 – файл RDP_tests.py

```
from recursive_descent_parser import RDParser


def test_my_grammar_1():
```

```
    cur_word = "a=a*b;"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert t


def test_my_grammar_2():
    cur_word = "a=b*('987654/cd);"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert t


def test_my_grammar_3():
    cur_word = "a=b*(\"1/cd)%#1010;"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert t


def test_my_grammar_4():
    cur_word = "a=b;"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert t


def test_my_grammar_5():
    cur_word = "a=a*!b;b=c+d;"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert t
```

```
def test_my_grammar_6():
    cur_word = "a=a;b=b;c=c;d=!!!d;e=#101010100101010;"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert t


def test_my_grammar_7():
    cur_word = "bc=ac+(af*('1010/b)+'111)-\"10;"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert t


def test_my_grammar_8():
    cur_word = "a=a*b"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert not t


def test_my_grammar_9():
    cur_word = "a=abc"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
        _ = e

    assert not t


def test_my_grammar_10():
    cur_word = "a=#9;"

    t = False

    try:
        t = RDParser.parse(cur_word)
    except Exception as e:
```

```
        _ = e
    assert not t
```