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

# Цель работы

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

# 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.

## 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*](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*

# Ход работы

*Часть 1*

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

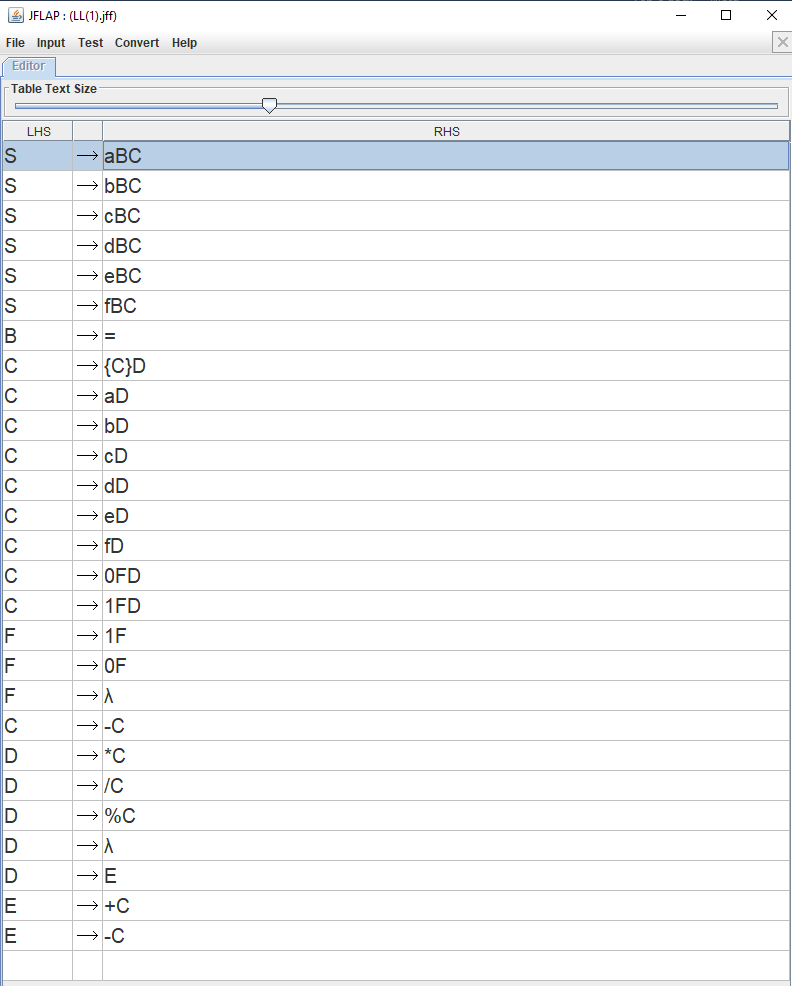


Рисунок 1 – полученная LL(1)-грамматика (файл LL(1)-1.jff)

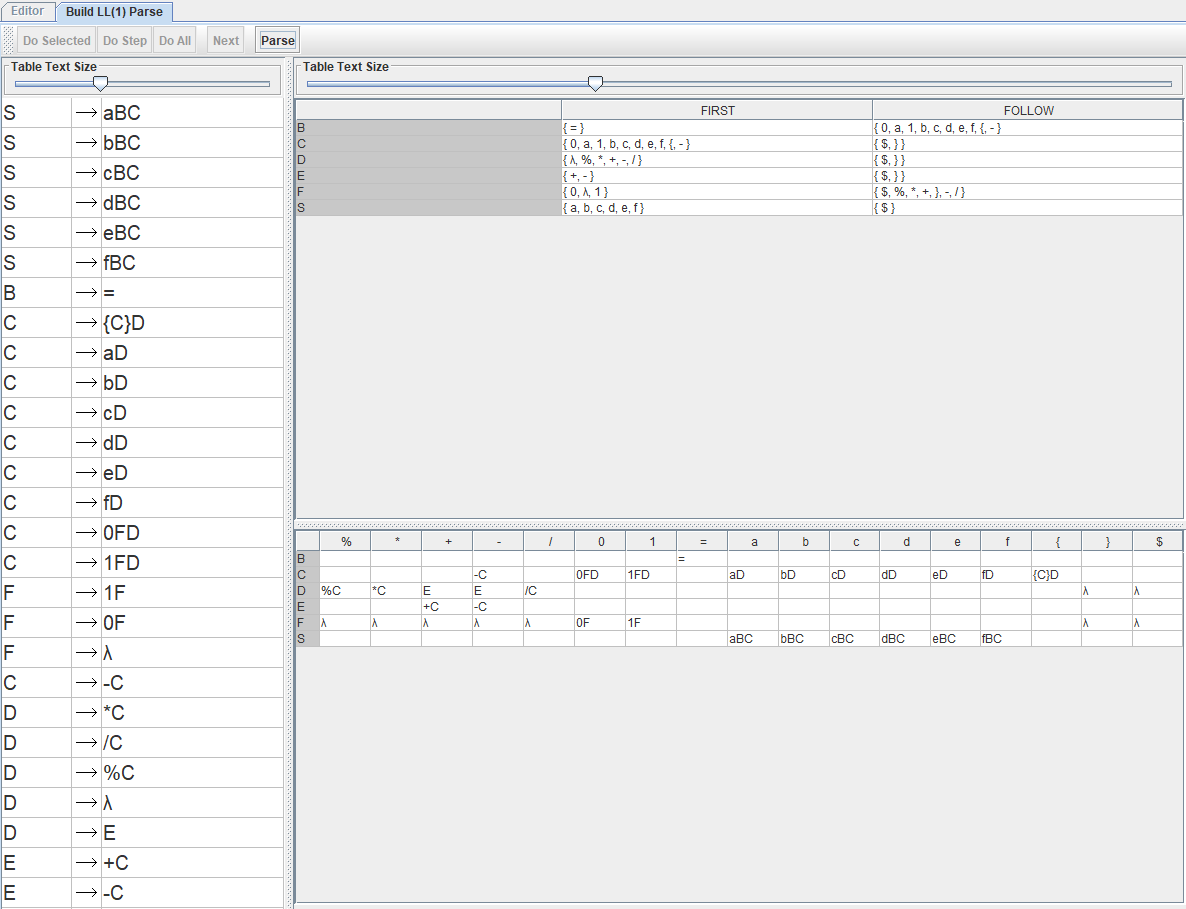


Рисунок 2 – таблица синтаксического LL(1)-анализа

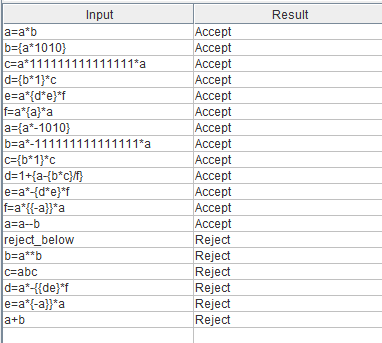


Рисунок 3 – Тестирование полученной грамматики

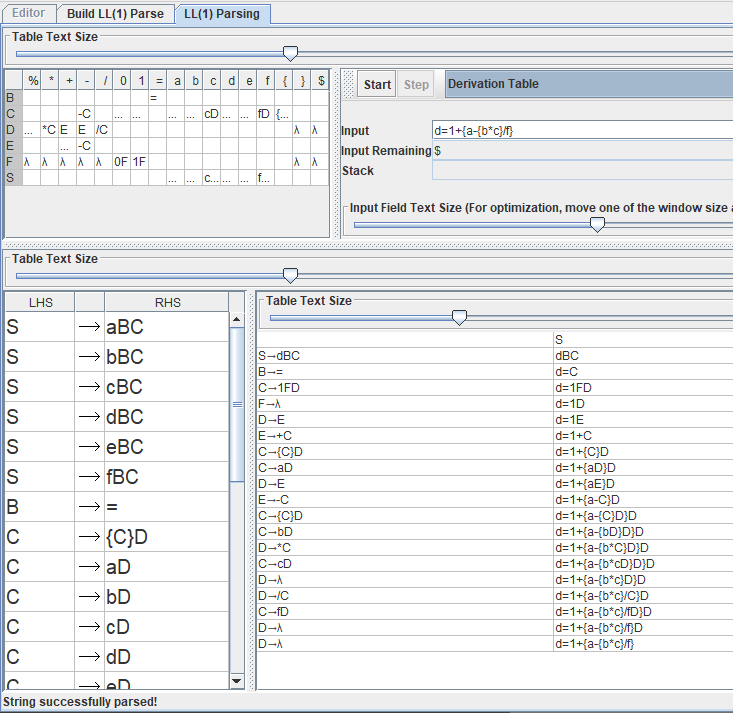


Рисунок 4 – Перехват экрана распознавания

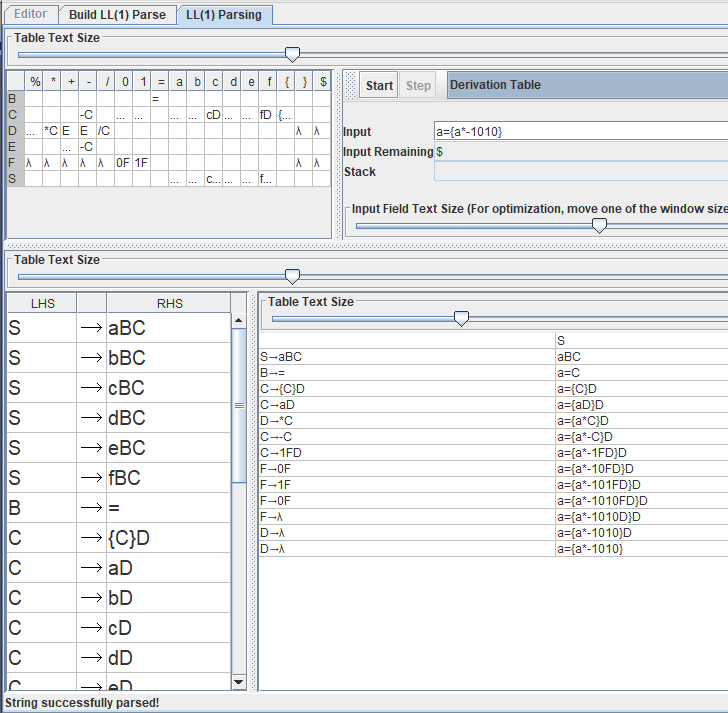


Рисунок 5 – Перехват экрана распознавания

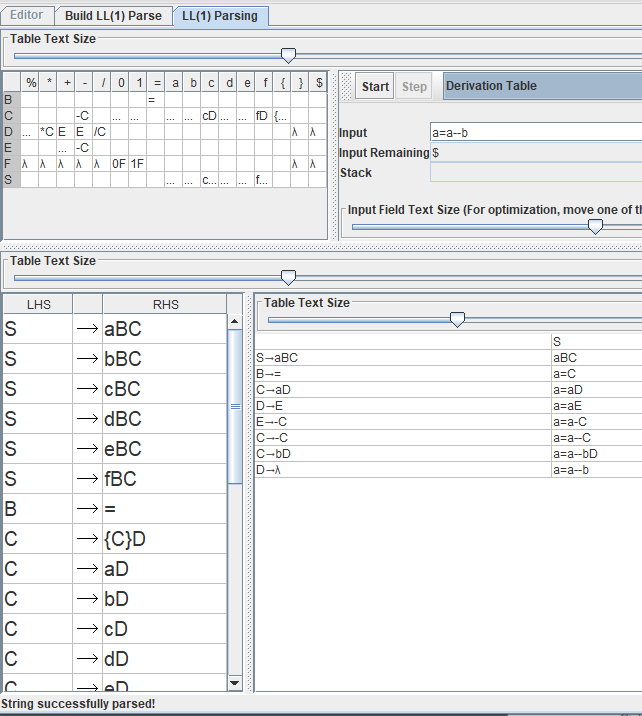


Рисунок 6 – Перехват экрана распознавания

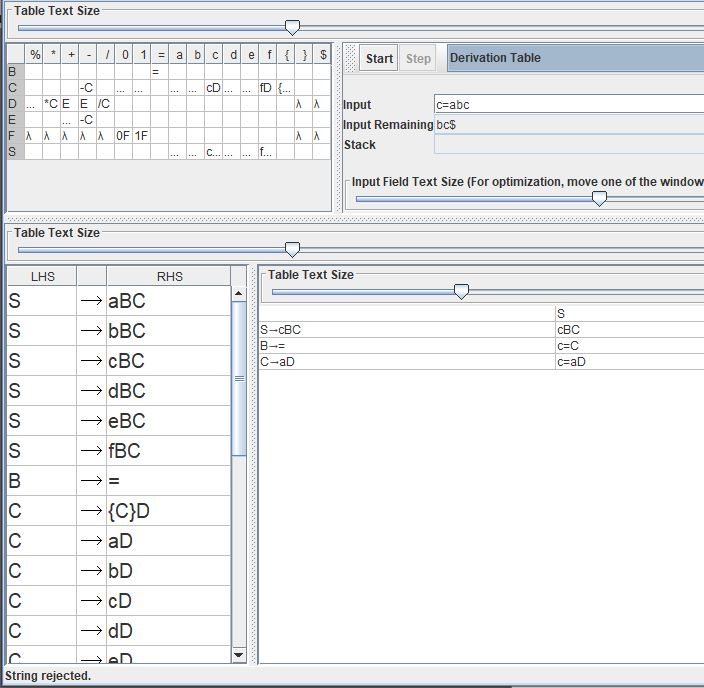


Рисунок 7 – Перехват экрана распознавания (для неверной строки)

*Часть 2*

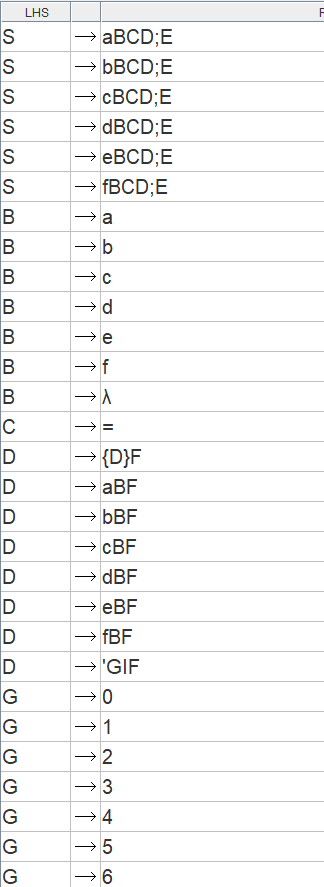


Рисунок 8 – полученная LL(1)-грамматика (файл LL(1)-2.jff; операция унарного минуса обозначена как «@»; идентификаторы систем счисления: «#» - 2, «”» - 8 «’» - 10)

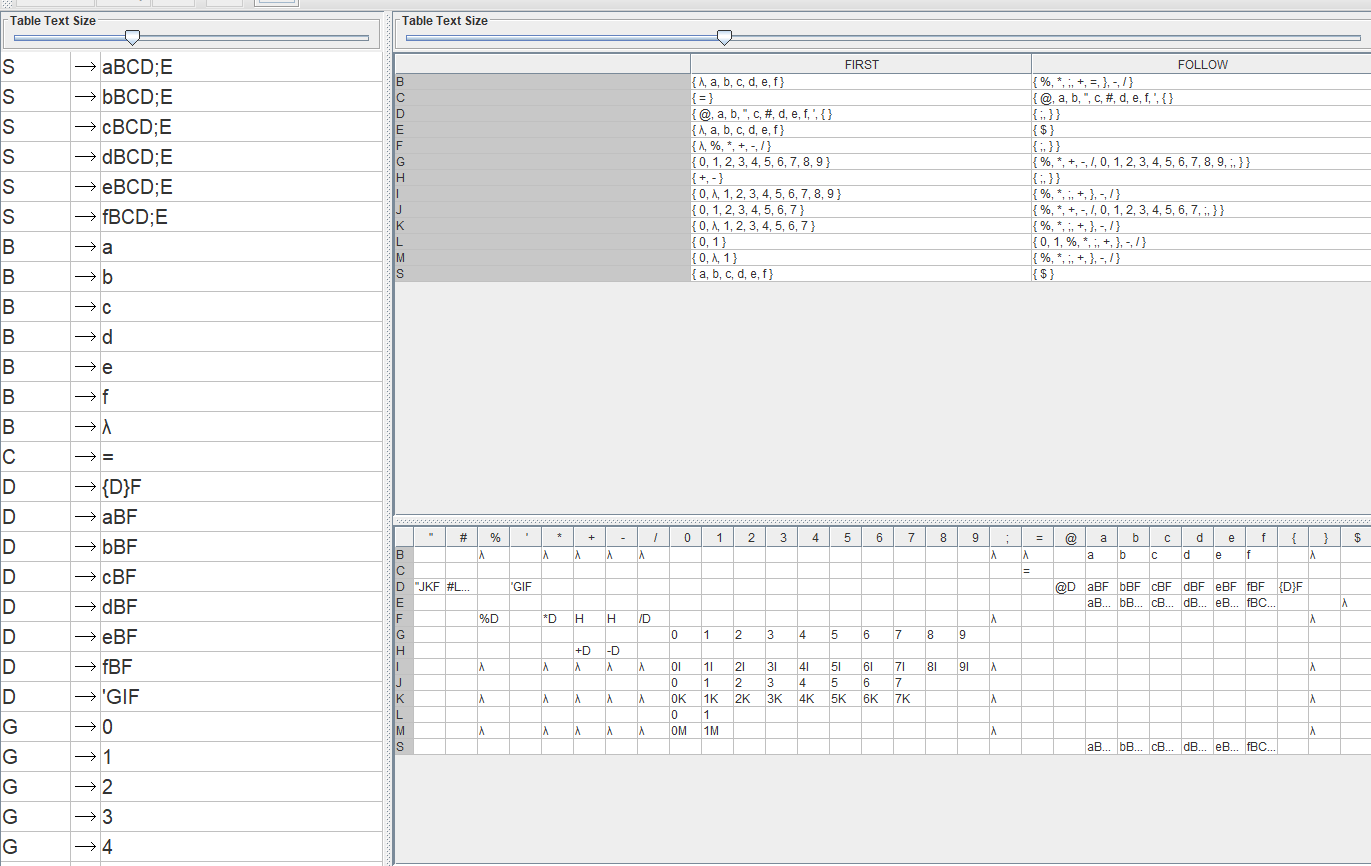
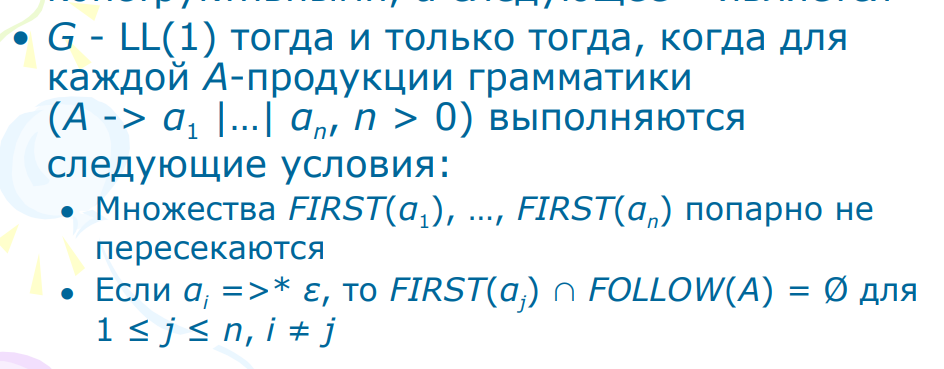


Рисунок 9 – Таблица СА

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



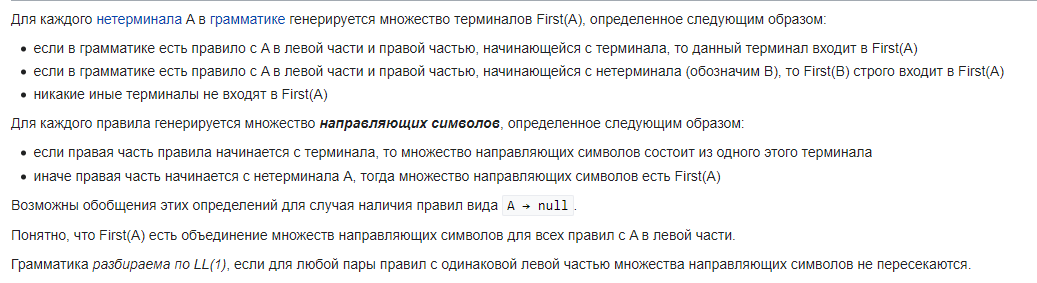


Рисунок 10 – Правила, которым удовлетворяет LL(1)-грамматика

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.

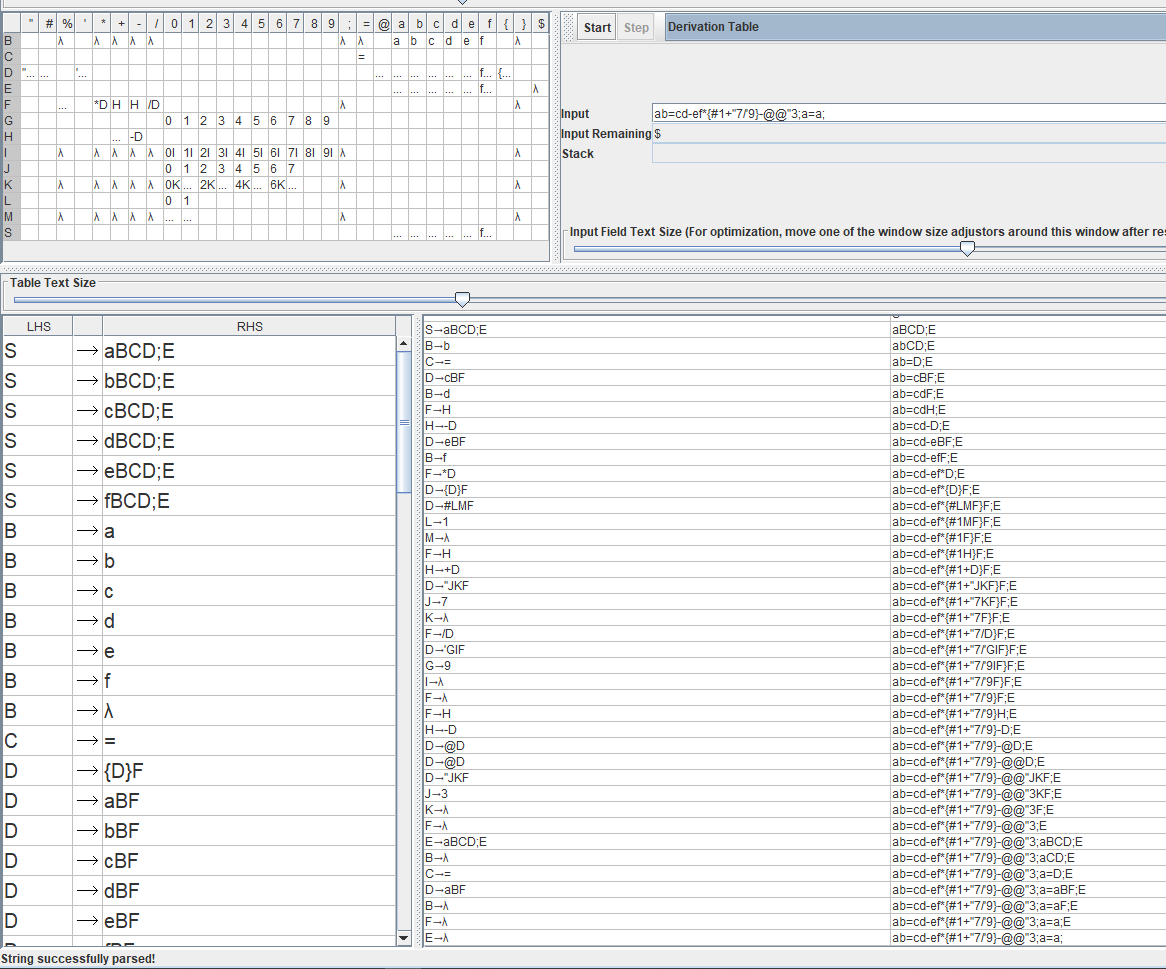


Рисунок 11 – Перехват экрана распознавания

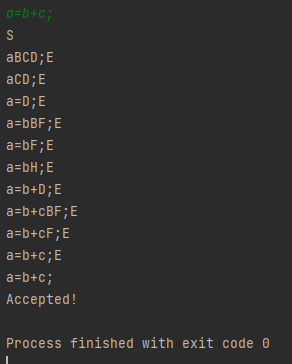


Рисунок 12 – Наглядное тестирование программной реализации

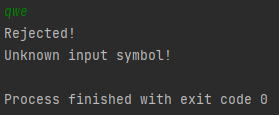


Рисунок 13 – Наглядное тестирование программной реализации

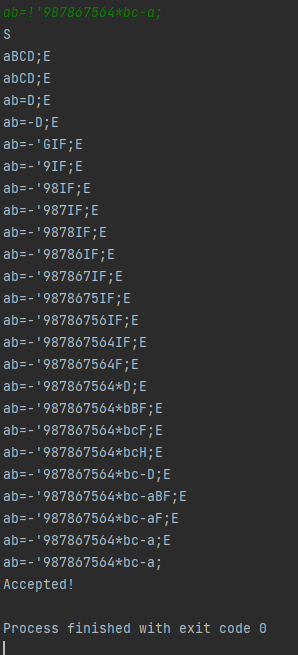


Рисунок 14 – Наглядное тестирование программной реализации

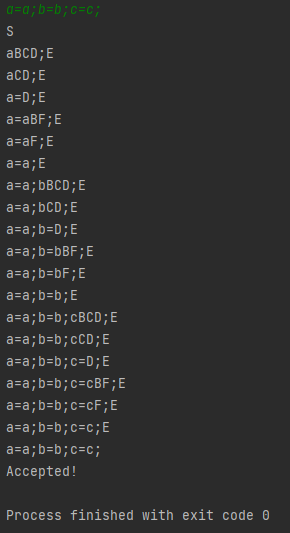


Рисунок 15 – Наглядное тестирование программной реализации

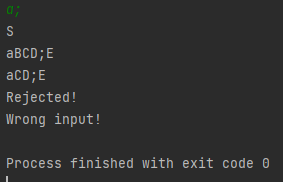


Рисунок 16 – Наглядное тестирование программной реализации

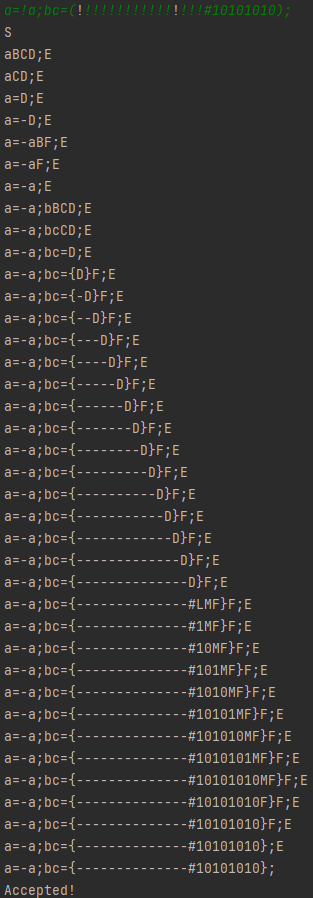


Рисунок 17 – Наглядное тестирование программной реализации

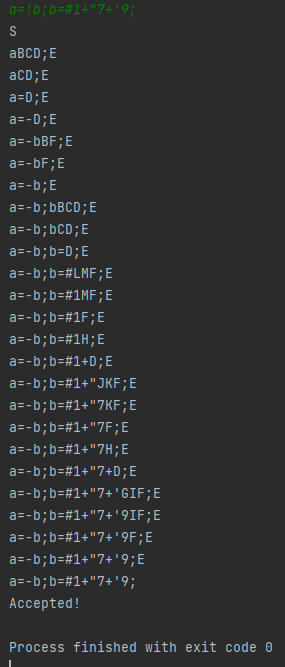


Рисунок 18 – Наглядное тестирование программной реализации

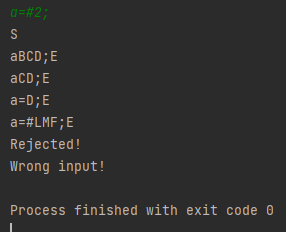


Рисунок 19 – Наглядное тестирование программной реализации

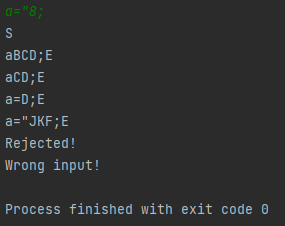


Рисунок 20 – Наглядное тестирование программной реализации

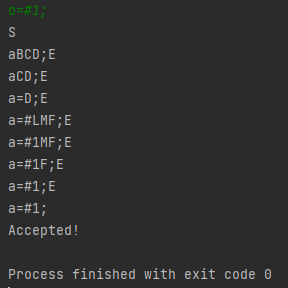


Рисунок 21 – Наглядное тестирование программной реализации

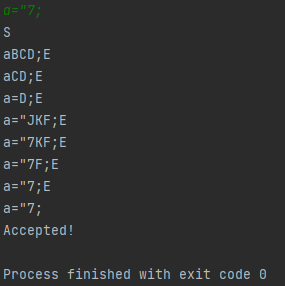


Рисунок 22 – Наглядное тестирование программной реализации

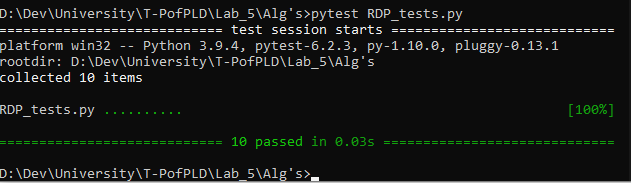


Рисунок 23 – Тестирование программной реализации с помощью набора тестов

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

*Часть 3*

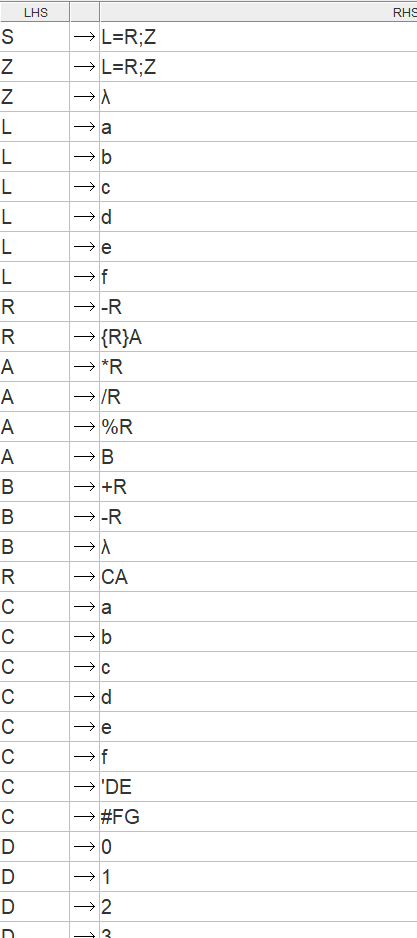


Рисунок 24 – Полученная грамматика для SLR анализа (SLR.jff; идентификаторы систем счисления: «#» - 2, «’» - 10)

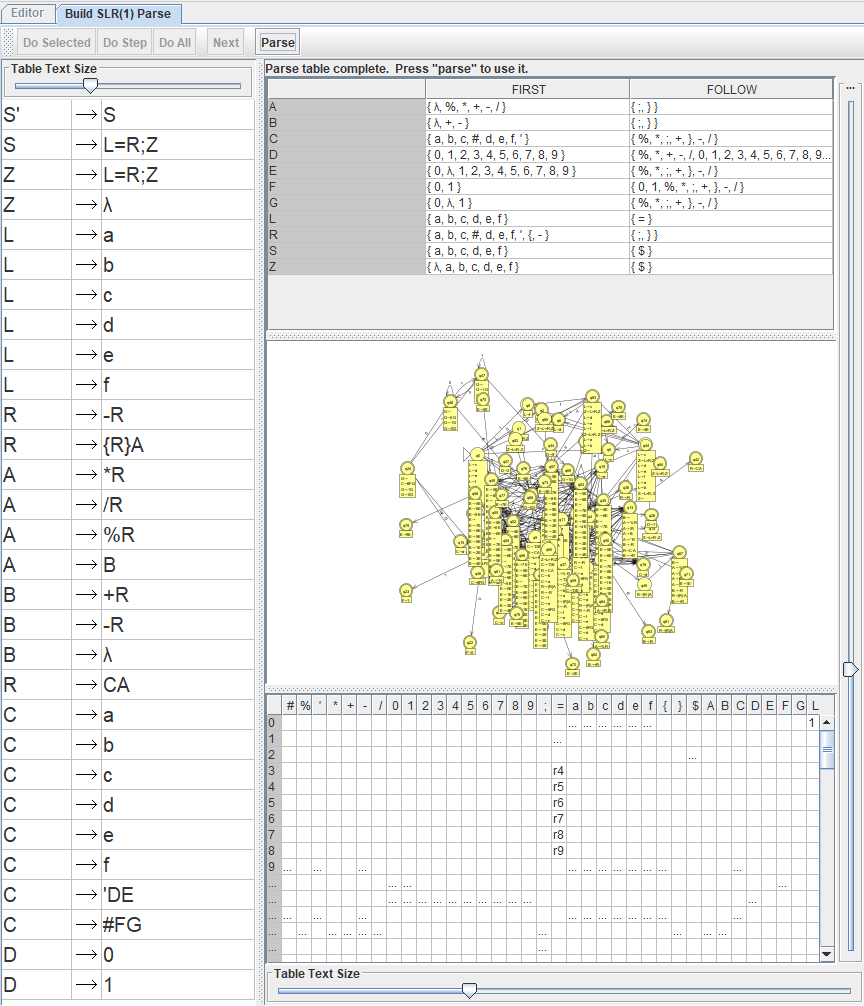


Рисунок 25 – Множества FIRST и FOLOW, ТСА (PDA.jff)

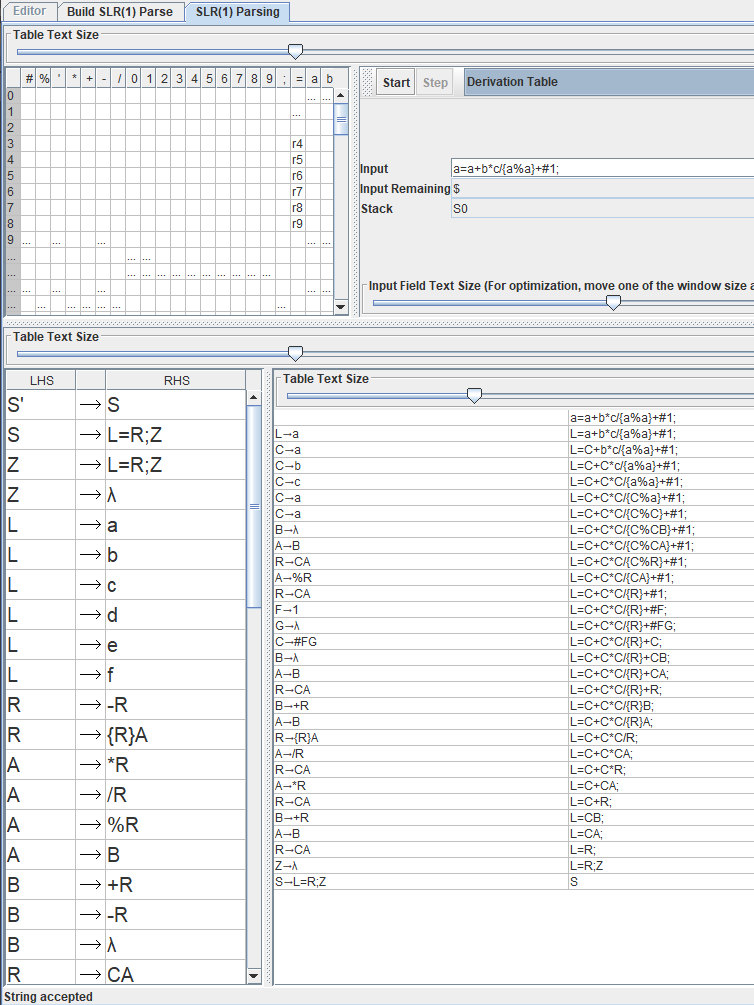


Рисунок 26 – Перехват экрана распознавания

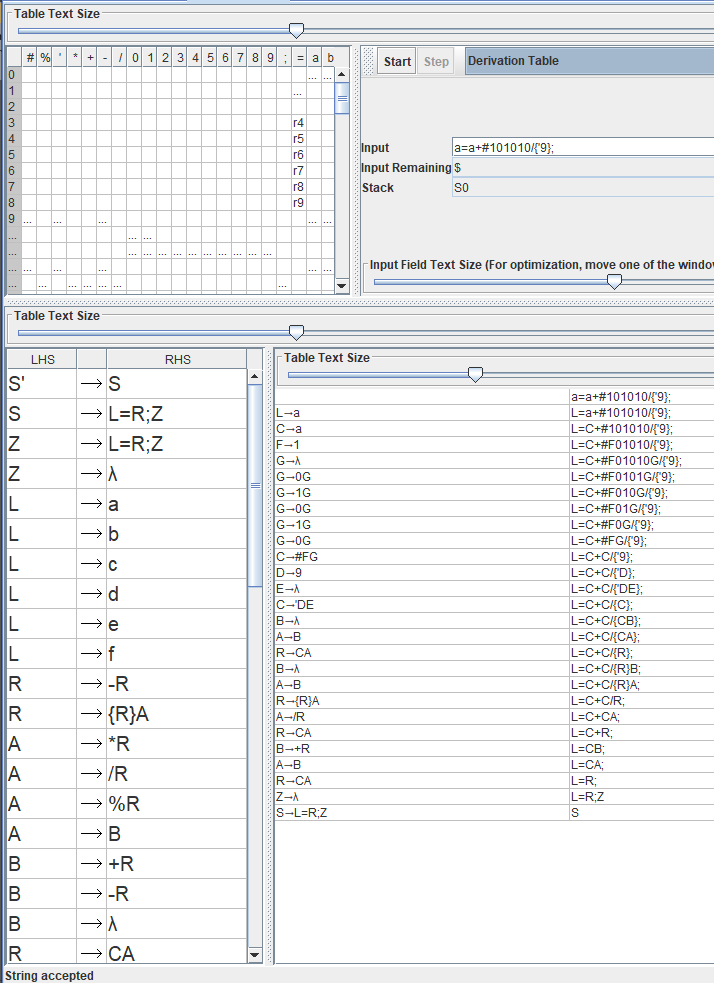


Рисунок 27 – Перехват экрана распознавания

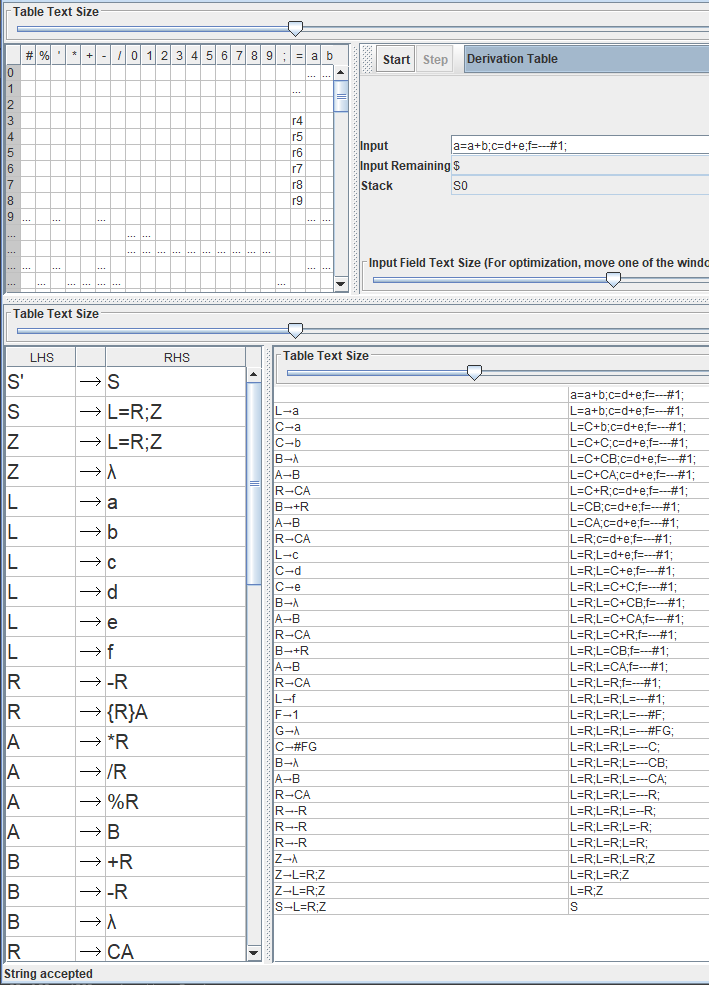


Рисунок 28 – Перехват экрана распознавания

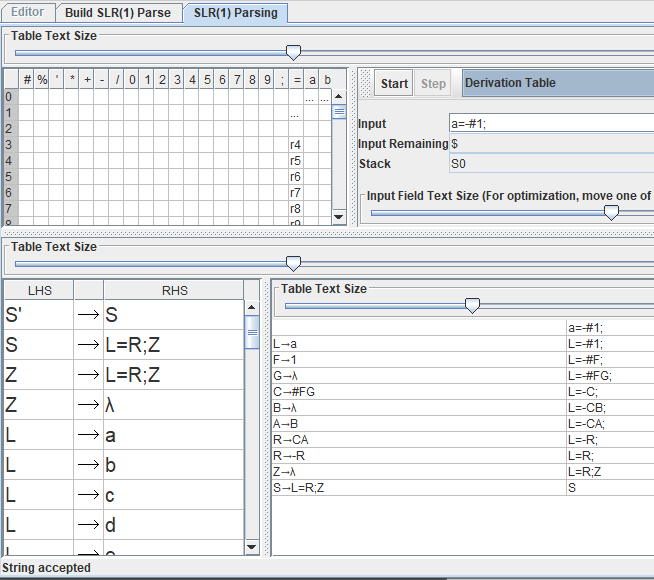


Рисунок 29 – Перехват экрана распознавания

# 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