Звіт
До лабораторної роботи 4
З дисципліни “Системне програмування”
Студентів 3 курсу ФКНК КНУ Тараса Шевченка
Групи МІ-31
Бідзілі Святослава та Наумця Захара

Постановка задачі

Командний проєкт, для 3-5 студентів у команді
Розробити LL(1)-синтаксичний аналізатор для заданої граматики, який будує AST або визначає та локалізує синтаксичну помилку:
1) запрограмувати всі необхідні функції: Firsk(k), Follow(k), побудова таблиці управління, власне аналізатор по таблиці
2) запрограмувати допоміжні функції: пошук епсилон-нетерміналів, читання і розбір введеної граматики, тощо
на додаткові бали:
3) LL(k) аналізатор для k>1 = +6 балів
4) також запрограмувати аналізатор методом рекурсивного спуску = +4 бали
5) реалізувати LALR-аналізатор (на прикладі граматики мови C) = +10 балів
6) візуалізація дерева виводу (AST) = +3 бали
На додаткові бали має бути саме реалізація (**розробка**, написання коду,
а **не використання готових** API або інших утиліт).
Кожний учасник команди має запрограмувати якусь частину проєкту, навіть менеджер проєкту. Розподілити ззадачі між членами команди +- порівну.

Виконання

Для вирішення поставленої задачі ми використовували мову python та створили застосунок використовуючи ООП. Було виконано пункти 1, 2, 3, 4, 6.

Застосунок створювався командою з двох учасників - Бідзіля Святослава та Наумця Захара.

Весь код проекту з невеликою документацією та прикладами роботи доступний за посиланням https://github.com/anakib1/MangoParsers

Основна структура

Всі елементи застосунку використовують базові сутності - Rule, Grammar.
Rule - це одне правило виведення в граматиці

Grammar - це граматика, яка складається з багатьох правил, має свій алфавіт, та зберігає наявність терміналів та нетерміналів.

Всі реалізовані парсери (наразі RecursiveParser та LLKParserWrapped) наслідують базовий клас IParser та мають однаковий інтерфейс, для спрощення роботи з збільшеним набором інструментів.

Також реалізовано клас Visualiser, який відтворює дерево парсингу для заданого рядка і граматики.

Деталі реалізації

grammar.py

Код написаний Святославом.

В файлі core/grammar.py розміщуються базові сутності - наприклад BaseSymbol - базовий клас для терміналів і нетерміналів.

```
class BaseSymbol:
    """
    Basic token. Could be Terminal or NonTerminal.
    """

    __slots__ = ['symbol']

    def __init__(self, symbol : str) -> None:
        if len(symbol) > 1:
            symbol = f'[{symbol}]'
        self.symbol = symbol

    def __repr__(self) -> str:
        return self.symbol

    def __eq__(self, other : object) -> bool:
        if not isinstance(other, BaseSymbol):
            return False
        return self.symbol == other.symbol

    def __hash__(self) -> int:
        return hash(self.symbol)
```

Додатково наявні два окремі класи для терміналів/нетерміналів:

```
class Terminal(BaseSymbol):
    """
    Symbols that are terminal
    """

    def __init__(self, symbol : str) -> None:
        assert symbol.islower() or not symbol.isalpha(),
'terminals MUST be lowercase'
        super().__init__(symbol)

    def isEpsilon(self):
        return self.symbol == "[eps]"

class NonTerminal(BaseSymbol):
    """
    Symbols that could produce rules
    """

    def __init__(self, symbol: str) -> None:
        assert symbol[0].isupper(), 'nonterminals should start
from uppercase characters'
        super().__init__(symbol)

    def isEpsilon(self):
        return False
```

Після створення базових класів для символів, створюються додатково клас для правила і граматики.

```
class Rule:
    """
    Rule of from A -> [Terminal|NonTerminal]*
    Flyweight pattern in use, we don't store references to
provided arguments, and
    you should not store references to returned values.
    """

    __slots__ = ['st', 'en']
```

Клас Rule виражає собою одне правило граматики, містить декілька допоможних функцій важливих в реалізації деяких парсерів, але не містить важливої логіки.

Наступний клас Grammar виражає цілісну сутність граматики - вона складається з набору правил, терміналів, нетерміналів, початкового символу,

```
class Grammar:
    vocab : Set[BaseSymbol]
    non_terms : Set[NonTerminal]
    terms : Set[Terminal]
    start_symbol : NonTerminal
    rules : Set[Rule]

    """
    Basic class for grammar entity.
    Flyweight pattern in use, we don't store references to
provided arguments, and
    you should not store references to returned values.
    """
```

В класі граматики наявна одна допоміжна дія - прибирання епссілон-правил. Це статичний метод, який повертає новий об’єкт класа Grammar з еквівалентною граматикою без наявних епсілон-правил.

```
    def remove_eps_rules(self):
        creatures = self.find_creatures()
        ret = Grammar(set(), set(), [])
        for rule in self.rules:
            creatures_in_rule = [x for x in range(len(rule.en)) if
rule.en[x] in creatures]
            ln = len(creatures_in_rule)
            for mask in range(1 << ln):
                nf = []
                for i in range(len(rule.en)):
                    if i in creatures_in_rule:
```

```
                    pos = creatures_in_rule.index(i)
                    if mask & 1 << pos:
                        nf.append(copy(rule.en[i]))
                else:
                    nf.append(copy(rule.en[i]))
            rule_to_add = Rule(copy(rule.st), nf)
            if not rule_to_add.isEpsilon(set()):
                ret.add_rule(rule_to_add)
    return ret
```

Також наявний метод для зчитування граматики з файлу, або масива рядків.

lookaheadUtils.py

Файл lookaheadUtils.py містить основні методи для створення допоміжних таблиць для LL(k) граматик. Це статичні методи, які зазвичай приймають граматику та повертають результат - таблицю. Код написаний Захаром.

```
def checkNonTerminalRule(rule: Rule) -> bool:
        """
        checks whether the end of the rule has only terminals
        in other words checks whethere the rule is terminal
        """
        for sym in rule.en:
            if isinstance(sym, NonTerminal):
                return False
        return True
```

Метод checkNonTerminalRule перевіряє чи є хочаб один нонтермінал в правилі.

Також наявний набір методів, які к-конкатенують декілька рядків, множин рядків або аналогічних сутностей.

```
     def KconcatenateTwostrings(str1: Tuple[Terminal], str2:
Tuple[Terminal], k: int) -> Tuple[Terminal]:

    def KconcatenateTwoSets(set1: Set[Tuple[Terminal]], set2:
Set[Tuple[Terminal]], k: int) -> Set[Tuple[Terminal]]:

    def KconcatenateListOfSets(lst: List[Set[Tuple[Terminal]]], k:
int) -> Set[Tuple[Terminal]]:
```

Далі, наявні самі методи, які вже створюють таблиці парсерів - стандартним способом з методичних матеріалів, а саме

```
    def firstK(grammar: Grammar, k: int) -> dict[NonTerminal,
Set[Tuple[Terminal]]]:
        """
        for every terminal in grammmar returns the set of all
        possible firstk tuples that the NonTerminal can produce
        """
```

Наступний метод знаходить first для послідовності символів, а не лише для нетерміналу, що зручно для подальшої реалізації.

```
    def sequenceFirstK(seq: Tuple[BaseSymbol], dct:
dict[NonTerminal, Set[Tuple[Terminal]]], k: int) ->
Set[Tuple[Terminal]]:
        """
        return every possible firstK tuples for sequence of
termainl and
        non-terminal symbols
        """
```

А також

```
def followK(grammar: Grammar, k: int):
        """
        for every terminal in grammmar returns the set of all
        possible firstk tuples that the NonTerminal can produce
        """
```

Цей метод знаходить множину follow для всіх нетерміналів граматики і заданого k.

llk.py

Файл llk.py містить реалізацію парсера для LL(k) граматик.

Код написаний Захаром

Для початку, в файлі є функція, яка знаходить таблицю керування для заданої граматики, використовуючи множини first, follow, які будуються за дпоомогою utils файлу.

```
def buildLLKTable(grammar: Grammar, k: int):
    """
    builds a table of rules for stack automata to parse strings
    this function returns table, reverse_ordering
    table : dict[NonTerminal, dict[Tuple[Terminal], int]]
    reverse_ordering : dict[int, Rule]
    where table is the table for transitions and reverse_ordering
is the
    labeling of rules for the table

    The function returns None if the grammar is not strong LL(k)
    """
```

Після побудови таблиці керування, дуже нескладно будувати сам парсер. Функція LLKParser проводить процес проходу по рядку згідно таблиці керування і повертає послідовність правил.

```
def LLKParser(string: str, grammar: Grammar, table:
Dict[NonTerminal, Dict[Tuple[Terminal], int]], ordering: Dict[int,
Rule], k: int) -> List[int]:
    """
    Parse the given string using LL(k) parser
    if the string can not be parsed prints the error and return
None
    otherwise returns the sequence of rules to produce the string
    """
```

Для сумісності з ООП дизайном, також файл містить “обгортку” цих функцій для створення об’єкту LLKParserWrapper

```
class LLKParserWrapped(IParser):

    def __init__(self, k = 5):
        self.k = k

    def init(self, grammar:Grammar) -> None:
        self.grammar = grammar
        self.table, self.order = buildLLKTable(grammar, self.k)

    def verify(self, s: str) -> bool:
        ret = LLKParser(s, self.grammar, self.table, self.order,
self.k)
        if ret is None:
            return False
        return True

    def parse(self, s : str) -> List[Rule]:
        ret = LLKParser(s, self.grammar, self.table, self.order,
self.k)
        if ret is None:
            return []
        return [self.order[x] for x in ret]
```

Зазначимо, що при ініціалізації парсера, необхідго передати значення k граматики, автоматичне розпізнавання не реалізовано.

RecursiveParser.py

Код написаний Святославом

В файлі RecursiveParser.py реалізовано простий рекурсивний парсер - він перебирає всі варіанти дій у випадках shift-reduce (reduce/reduce) conflicts і знаходить будь-яку послідовність правил, яка приводить до правильного результату.

В проміжних функціях ми перевіряємо, чи можемо ми виконати поточну операцію, та виконуємо операції.

```
    def __init__(self, grammar = None):
        if grammar is not None:
            self.init(grammar)

    def init(self, grammar : Grammar):
        self.grammar = grammar.remove_eps_rules()
        self.rules_by_symbol = {}
        for rule in self.grammar.rules:
            self.rules_by_symbol[rule.st] =
self.rules_by_symbol.setdefault(rule.st, []) + [rule]


    def is_compatible(self, rule : Rule):
        if len(rule.en) > len(self.parse_stack):
```

```
            return False
        for i,x in enumerate(rule.en[::-1]):
            if self.parse_stack[-1-i] != x:
                return False
        return True

    def reduce(self, rule:Rule):
        self.parse_stack = self.parse_stack[:-len(rule.en)]
        self.parse_stack.append(rule.st)
```

В функції try_parse ми просто на кожному кроці знаходимо всі можливі операції і пробуємо по черзі їх виконати, для знаходження будь-якої послідовності правильних дій. При конфлікті або “неправильно” вибраній операції, ми відкатуємо стан перебору на минулий.

```
def try_parse(self) -> bool:
        if len(self.input) == 0 and len(self.parse_stack) == 1 and
self.parse_stack[0] == self.grammar.start_symbol:
            return True

        moves = []
        for rule in self.grammar.rules:
            if self.is_compatible(rule):
                moves.append((1, rule))
        if len(self.input) > 0:
            moves.append((2, 'shift'))

        for move in moves:
            saved_moment = deepcopy((self.parse_stack,
self.input))
            if move[0] == 2:

self.parse_stack.append(SymbolUtils.getSymbol(self.input[0]))
                self.input = self.input[1:]
                if (self.try_parse()):
                    return True
            else:
                self.reduce(move[1])
                sz = len(self.move_stack)
                if (self.try_parse()):
                    self.move_stack = self.move_stack[:sz]  +
[deepcopy(move[1])] + self.move_stack[sz:]
                    return True

            self.parse_stack, self.input = saved_moment

        return False
```

Drawer.py

Код написано Святославом
Цей клас створено для візуалізації виводу з парсерів у вигляді дерева парсингу.

Для облегшення малювання самого графу було використано лібу graphviz

В классі Visualiser наявні 3 функції - add (додає нову вершину до дот-об'єкту графа, правильно форматуючи її назву)

```
    def add(self, val : BaseSymbol) -> int:

        name = str(val)
        clr = None
        if isinstance(val, Terminal):
            self.terminal_counter += 1
            name = "'" + name + "'" +  ' (' +
str(self.terminal_counter) + ')'
            clr = 'Red'

        self.dot.node(str(self.counter), name, color = clr)
        self.counter += 1
        return val, self.counter - 1
```

build_tree, яка рекурсивно намагається будувати дерево, виконуючи правила один за одним зліва направо.

```
def build_tree(self, vertex : Tuple[BaseSymbol, int]) -> int:

        if isinstance(vertex[0], Terminal) :
            return vertex[1]

        if len(self.order) == 0:
            return
        rule = self.order.pop(0)
        for x in rule.en:
            self.dot.edge(str(vertex[1]),
str(self.build_tree(self.add(x))))

        return vertex[1]
```

draw - яка малює саме дерево з переданого їй порядку правил.

```
    def draw(self, order : List[Rule]):
        self.counter = 0
        self.terminal_counter = 0
        self.order = order
        self.dot = graphviz.Digraph('parsing result')
        self.build_tree(self.add(order[0].st))

        self.dot.render('dot', format='png', view=True)
```

Тести

Всі тести написані Святославом

Для перевірок роботоздатності коду було використано бібліотеку unittest та створенно декілька тестів як на парсери, так і на таблиці first і follow.

Всі тести парсера мають приблизно однаковий вигляд - ми створюємо граматику, як набір правил, після чого ініціалізуємо парсер, і перверяємо що verify повертає очікуване значення.

Через використання ООП, ми можемо легко тестувати різні види парсерів, замінивши LLKParserWrapped(3) на RecursiveParser, що і зроблено в його тестах.

```
def testParserLL3(self):
        grammar = Grammar.read(['S -> aSA', 'S -> eps', 'A -> aabS
| c'])
        parser = LLKParserWrapped(3)
        parser.init(grammar)

        self.assertTrue(parser.verify('aaab'))
        self.assertTrue(parser.verify('aaabaaab'))
        self.assertTrue(parser.verify('aaabaaabac'))
```

Для тестів таблиць, використано аналогічний підхід. Створюємо таблицю і перевіряємо її корректність

```
class LookaheadUtilsTest(unittest.TestCase):
    def checkSets(self, a : Set, b : Set):
        self.assertEqual(len(a), len(b))
        for x in a:
            self.assertTrue(x in b, f'set b does not contain
element {x}')

    def testFirstKTable(self) :

        grammar = Grammar.read(['S -> BA', 'A -> +BA | eps', 'B ->
DC', 'C -> *DC | eps', 'D -> (S) | a'])

        result = lookaheadUtils.firstK(grammar, 2)
        self.checkSets(
            result[SymbolUtils.getSymbol('A')],
            set([
                SymbolUtils.getSymbols('eps'),
                SymbolUtils.getSymbols('+a'),
                SymbolUtils.getSymbols('+(')
            ])
        )
        self.assertTrue(True)
```

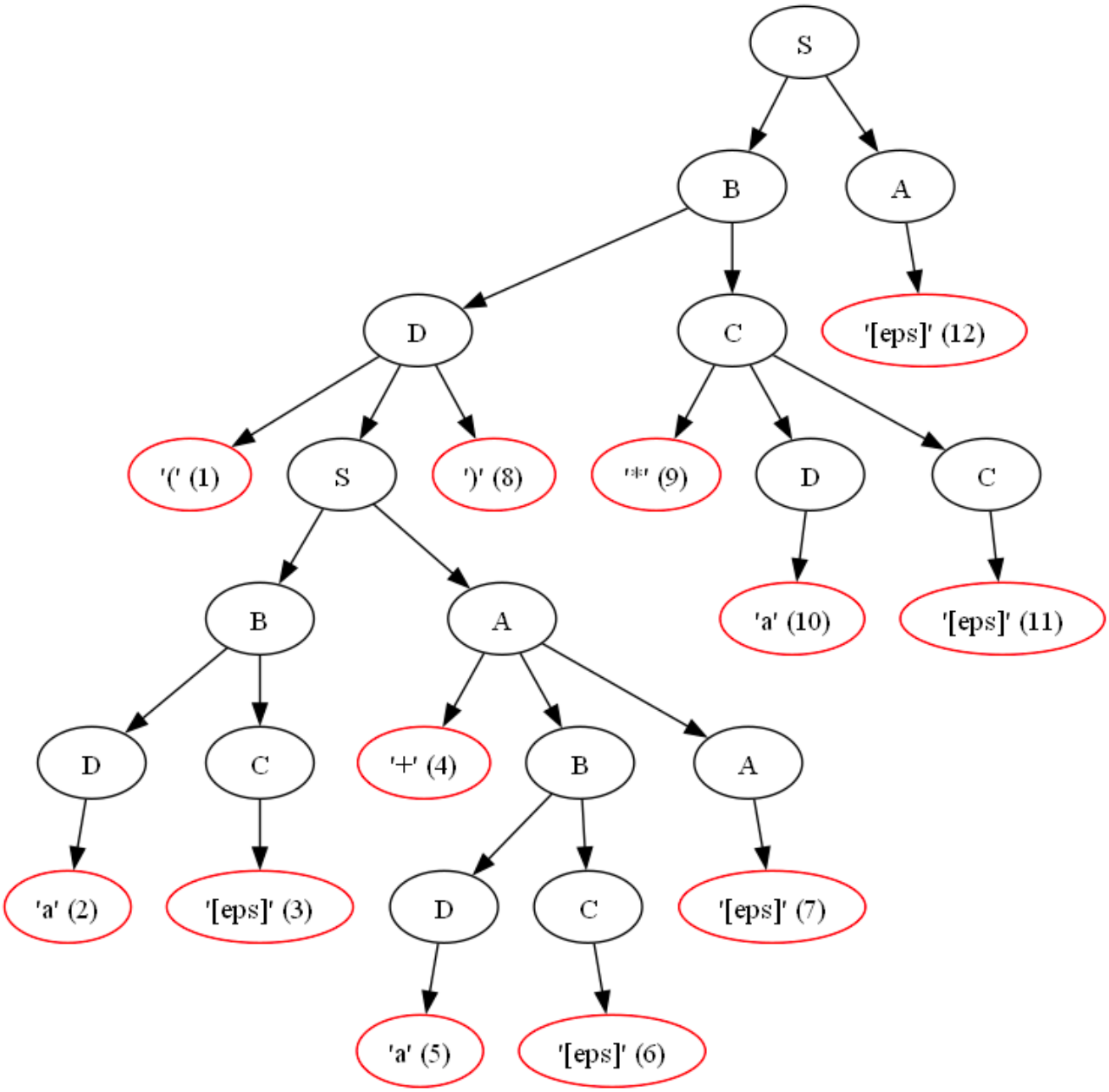
Окремо можна протестувати виведення дерева парсингу, оскільки складно це зробити автоматично. В файлі drawerTest.py просто запускається дерево парсингу при створеній граматиці.

Приклади роботи

Приклад дерева виводу для граматики

```
S -> BA,
A -> +BA | eps,
B -> DC,
C -> *DC | eps,
D -> (S) | a
```

Та слова `(a+a)*a`



При виклику парсера для невивідного слова (нарпиклад “a(“) ми отримаємо такий аутпут:

```
Synthax error near 1 symbol, can not parse C -> ((,)
[]
```

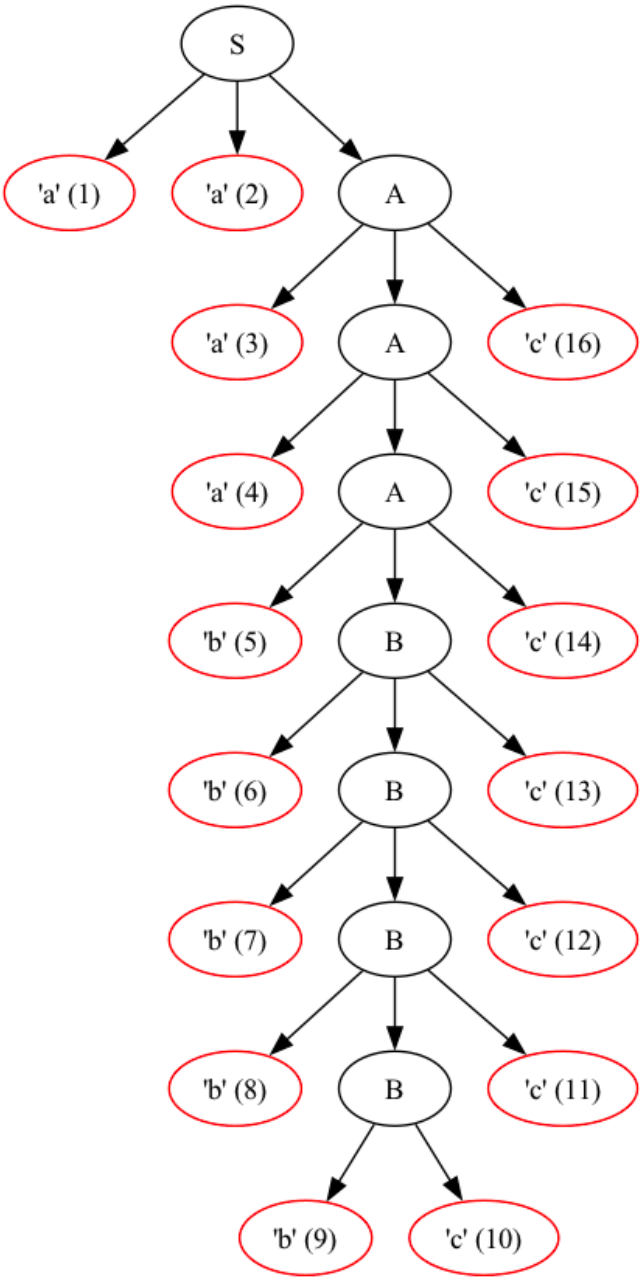
Парсер детально повідомляє про помилку, і повертає порожній масив правил.

На іншій (LL(2) граматиці) отримаємо такий вивід послідовності правил.

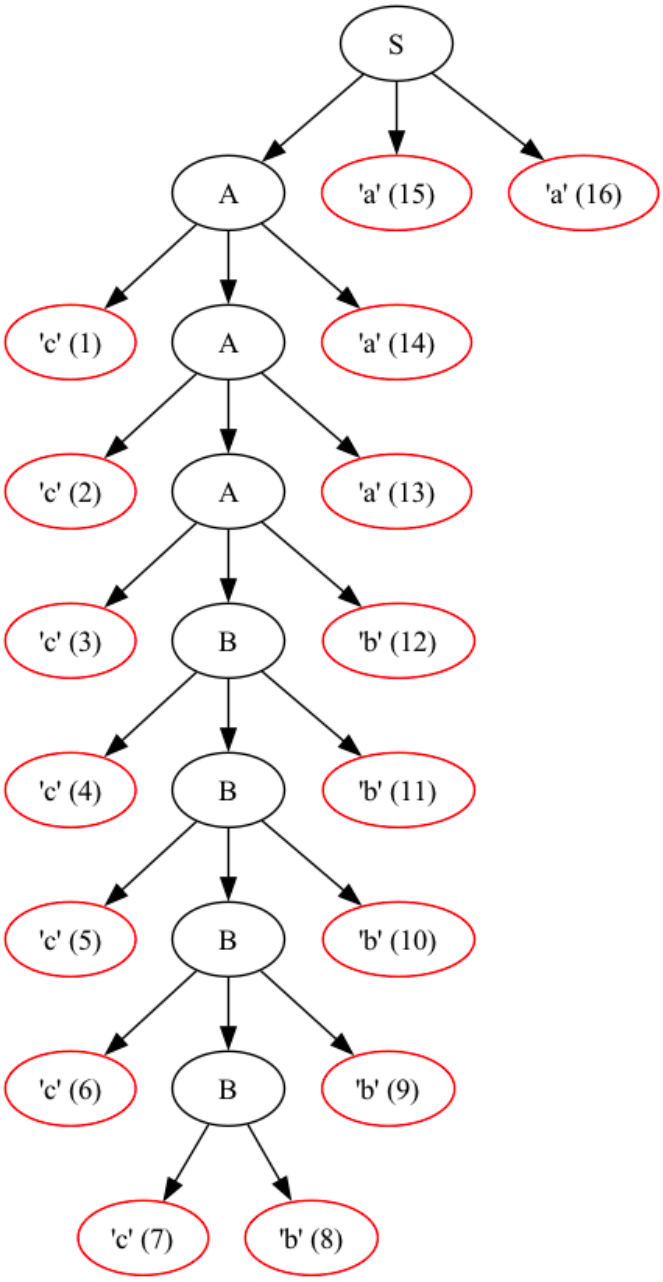
```
S -> aaA,
A -> aAc | bBc,
B -> bBc | bc

Word = ‘aaaabbbbbccccccc'
```

Output =

[S -> ['a', 'a', 'A'], A -> ['a', 'A', 'c'], A -> ['a', 'A', 'c'], A -> ['b', 'B', 'c'], B -> ['b', 'B', 'c'], B -> ['b', 'B', 'c'], B -> ['b', 'B', 'c'], B -> ['b', ‘c']]



Аналогічно, для рекурсивного парсеру отримаємо також вивід (але правила інвертуються зправа наліво, для побудови візуалізації) і візуалізацію дерева



Output =

[S -> ['A', 'a', 'a'], A -> ['c', 'A', 'a'], A -> ['c', 'A', 'a'], A -> ['c', 'B', 'b'], B -> ['c', 'B', 'b'], B -> ['c', 'B', 'b'], B -> ['c', 'B', 'b'], B -> ['c', ‘b’]]

Висновок

Під час виконання лабораторної роботи ми змогли реалізувати достаьньо потужні інструменти перевірки вивідності слів граматики, які можна реалізовувати вже навіть для простих мов програмування, та на практці затвердити теоретичні знання отримані на курсі.