**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

# ЗВІТ

# З ЛАБОРАТОРНОЇ РОБОТИ №4

з дисципліни «Системне програмування» на тему:

«LL(1)-синтаксичний аналізатор»

Студента 3 курсу, групи МІ-31

спеціальності 122 «Комп’ютерні науки»

Лихопуда О. Ю.

Викладач Поліщук Н. В.

Київ –2023

## ПОСТАНОВКА ЗАДАЧІ

Розробити LL(1)-синтаксичний аналізатор для заданої граматики, який будує AST або визначає та локалізує синтаксичну помилку:

1) запрограмувати всі необхідні функції: First(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).

## МЕТА

Метою цієї лабораторної роботи є розробка LL(1)-синтаксичного аналізатора для заданої граматики. Цей процес включає в себе так ключові аспекти:

Розуміння концепцій та принципів, які лежать в основі LL(1) граматик.

Набуття практичних навичок у розробці синтаксичних аналізаторів, розуміння принципів роботи з граматиками та оволодіння техніками обробки мовних конструкцій. Реалізація даних завдань сприятиме глибшому розумінню теорії компіляторів та інтерпретаторів.

## ОСНОВНІ ПРИНЦИПИ ВИКОНАННЯ ЛАБОРАТОРНОЇ

Визначити нетермінали, термінали та правила виводу, знайти епсілон-нетермінали. Розробити функції First(k) і Follow(k) для визначення множин слів, які можуть з’являтися на початку виведених рядків з нетермінала і які можуть безпосередньо слідувати за вказаним нетерміналом. Створити

таблицю управління, яка відображає всі можливі стани аналізатора. Реалізувати аналізатор рекурсивного спуску та розробити спосіб візуалізації абстрактного синтаксичного дерева. Робота виконуватиметься мовою програмування Python.

## ХІД ЛАБОРАТОРНОЇ РОБОТИ

Лабораторна виконувалась лише мною, а не командою. Спочатку потрібно створити клас для зберігання терміналів. У терміналів буде лише одне поле – text, але декілька методів – перевірка на те, чи є цей символ пустим (ε) та перевантаження методів для представлення екземпляру класа, як рядка, операцію порівняння та хешування:

```
class Terminal:
    oyl04
    def __init__(self, text):
        self.text = text
        if self.is_empty():  # Replace the text with 'eps' if it represents an empty terminal
            self.text = 'ε'

    # Check if the terminal symbol is an epsilon (empty symbol).
    9 usages (5 dynamic)  oyl04
    def is_empty(self):
        return self.text in ('ε', 'epsilon', 'eps')

    oyl04
    def __str__(self) -> str:
        return self.text

    oyl04
    def __repr__(self) -> str:
        return self.text

    oyl04
    def __eq__(self, o: object) -> bool:
        return self.text == o.text

    oyl04
    def __hash__(self) -> int:
        return hash(self.text)
```

Клас нетерміналів буде мати схожий вигляд, але замість обробки ε, потрібно зберігати масив, що може виводитися з цього нетермінала:

```python
class NonTerminal:
    oyl04
    def __init__(self, text):
        self.text = text
        # A list to hold production productions for this non-terminal.
        self.productions = []

    oyl04
    def __str__(self) -> str:
        return self.text

    oyl04
    def __repr__(self) -> str:
        return self.text

    oyl04
    def __eq__(self, o: object) -> bool:
        return o.text == self.text

    oyl04
    def __hash__(self) -> int:
        return hash(self.text)
```

Наступним буде клас граматики, спочатку ініціалізація. В ньому ініціалізуються змінні для зберігання регулярного виразу, терміналів, нетерміналів, знаходження виведень, епсилон-нетерміналів.

```python
class Grammar:

    oyl04
    def __init__(self, productions):
        self.pattern = None
        self.terminals = []
        self.non_terminals = []
        # Process the given productions to populate terminals and non-terminals.
        self.get_productions(productions)
        self.productions = productions
        # Identify non-terminals that can produce an epsilon (empty string).
        self.nullable_non_terminals = self.find_epsilon_producing_non_terminals()
```

Знаходження термінала, який є епсилоном:

```python
def get_epsilon(self):
    # Find a terminal representing an epsilon.
    eps = next((t for t in self.terminals if t.is_empty()), None)
    if eps is None:
        eps = Terminal('eps')
    return eps
```

Створення регулярного виразу для ідентифікації терміналів і нетерміналів:

```python
2 usages  oyl04
@staticmethod
# Generate a regex pattern to identify terminals and non-terminals in text.
def get_pattern(text_set):
    return f"({'|'.join(sorted([re.escape(t.text) for t in text_set], key=lambda x: len(x), reverse=True))})"
```

Функція, що шукає нетермінали, які можуть вивести епсилон. Для цього потрібно спочатку знайти ті нетермінали, які виводять епсилон напряму. Після цього потрібно ітераційно додавати в множину нові нетермінали, з яких можна вивести епсилон. Якщо є виведення пустого рядка (епсилона) для нетермінала, якого ще не додали в множину, то додаємо його. Ітерації закінчуться, коли множина на попередньому кроці буде рівна множині на теперішньому кроці:

```python
def find_epsilon_producing_non_terminals(self):
    nullable_non_terminals = set()
    # Check each non-terminal's production productions.
    for nt in self.non_terminals:
        for production in nt.productions:
            # If any production of a non-terminal directly produces an epsilon, add it to the set.
            if any(isinstance(symbol, Terminal) and symbol.is_empty() for symbol in production):
                nullable_non_terminals.add(nt)
                break

    # Continuously check for non-terminals that can indirectly produce an epsilon.
    changed = True
    while changed:
        changed = False
        for nt in self.non_terminals:
            if nt in nullable_non_terminals:
                continue
            for production in nt.productions:
                # If all symbols in a production are epsilon producers or empty terminals, add the non-terminal.
                if all(symbol in nullable_non_terminals or (isinstance(symbol, Terminal) and symbol.is_empty()) for
                       symbol in production):
                    nullable_non_terminals.add(nt)
                    changed = True
                    break

    return nullable_non_terminals
```

Функція get_production призначена для обробки виведень граматики.

Спочатку функція додає нетермінали, визначені у виведеннях, до списку нетерміналів граматики. Це досягається за допомогою спискового включення, яке перетворює кожен нетермінал на об'єкт NonTerminal. Далі створюється регулярний вираз для ідентифікації нетерміналів. Функція знаходить термінали в граматиці. Вона розбиває кожне виведення правила, фільтрує нетермінали за допомогою регулярного виразу та виділяє термінали, які потім додаються до множини. Для кожного ідентифікованого терміналу створюється екземпляр класу Terminal. Подібно до нетерміналів, створюється регулярний вираз для терміналів. Регулярні вирази терміналів та нетерміналів об'єднуються в один вираз. Цей вираз є важливим для ефективного розбору виведень. В кінці функція обробляє кожне виведення для кожного нетерміналу. Вона виявляє відповідності шаблонів у виведеннях і класифікує їх як термінал або нетермінал, додаючи їх до відповідного списку виведень.

```
def get_productions(self, productions):
    # Add non-terminals defined in the productions to the grammar's non_terminals list.
    self.non_terminals += [NonTerminal(n) for n in list(productions.keys())]
    # Create a regular expression pattern to identify non-terminals.
    n_pattern = Grammar.get_pattern(self.non_terminals)

    terminals = set()
    for n in self.non_terminals:
        for nt_production in productions[n.text]:
            # Split each rule production and identify terminals.
            g = nt_production.split()
            # Add identified terminals to the set.
            t = [re.sub(n_pattern, repl: ' ', p).split() for p in g]
            terminals |= set(itertools.chain.from_iterable(t))
    # Create Terminal objects for each identified terminal symbol.
    self.terminals = [Terminal(term) for term in terminals]
    # Print the identified terminals and non-terminals.
    print(self.terminals, self.non_terminals)

    # Create a regular expression pattern to identify terminals.
    t_pattern = Grammar.get_pattern(self.terminals)

    # Combine the terminal and non-terminal patterns into a single pattern.
    nt_pattern = f"{n_pattern}|{t_pattern}"
    self.pattern = nt_pattern

    # Process each production for each non-terminal and store in their 'productions' attribute.
    for n in self.non_terminals:
        for nt_production in productions[n.text]:
            n.productions.append([])
            for m in re.finditer(nt_pattern, nt_production):
                # Depending on whether the match is a terminal or non-terminal, add it to the production.
                if m.group(1):
                    n.productions[-1].append(next((nt for nt in self.non_terminals if nt.text == m.group(1))))
                elif m.group(2):
                    n.productions[-1].append(next((t for t in self.terminals if t.text == m.group(2))))
```

Також потрібно реалізувати функцію, яка буде читати граматику з файлу.

Нехай правила мають вигляд X -> P1 | P2 | P3 … | Pn, тоді потрібно розбити правило на дві частини, перша буде нетерміналом X, а друга виведеннями P1 | P2 | P3 … | Pn. Після чого розіб'ємо виведення по символу '|', що дасть змогу записати всі виведення для даного нетерміналу X:

```
@staticmethod
def read_grammar_from_file(file_path):
    productions = {}

    with open(file_path, 'r') as file:
        for line in file:
            # Split each line at the '->' symbol to separate the left and right parts of the rule production.
            parts = line.strip().split('->')
            if len(parts) == 2:
                left, right = parts
                # Trim whitespace from the left-hand side (non-terminal) of the production.
                left = left.strip()
                # Split the right-hand side of the rule into individual productions using '|' as the delimiter.
                right_productions = right.strip().split("|")
                # If the non-terminal is not already in the productions dictionary, add it with its productions.
                # Otherwise, extend its list of productions with the new ones.
                if left not in productions:
                    productions[left] = right_productions
                else:
                    productions[left].extend(right_productions)

    for key in productions.keys():
        productions[key] = ['ε' if prod == 'epsilon' else prod for prod in productions[key]]

    return productions
```

За допомогою функції re.finditer, функція перебирає всі відповідності у тексті, які відповідають шаблону self.pattern.

Якщо знайдена відповідність є нетерміналом / терміналом (знайдено в m.group(1) і m.group(2) відповідно), функція шукає відповідний об'єкт NonTerminal / Terminal у списку self.non_terminals / self.terminals та додає його до результату. Ця функція забезпечує мапінг між текстом та структурними елементами синтаксичного аналізатора:

```
def get_tnt_string(self, text):
    result = []
    for m in re.finditer(self.pattern, text):
        if m.group(1):
            # Append the corresponding non-terminal object to the result
            result.append(next((nt for nt in self.non_terminals if nt.text == m.group(1))))
        elif m.group(2):
            # Append the corresponding terminal object to the result
            result.append(next((t for t in self.terminals if t.text == m.group(2))))
    return result
```

Тепер потрібно запрограмувати функції для пошуку First(k), Follow(k), конкатенації та знаходження можливих рядків в класі FirstFollow.

Функція compute_first_k реалізує алгоритм обчислення множини First_k для кожного нетермінала в граматиці. Ця множина містить усі можливі початкові рядки довжини до k, які можуть бути отримані з цього нетермінала. Спочатку для кожного нетермінала в граматиці створюється порожня множина. Функція виконує ітераційний процес, де на кожній ітерації обчислюється First_k для кожного нетермінала на основі його виведень та вже визначених First_k для інших нетерміналів. Перед кожною ітерацією створюється копія поточного стану множини first, щоб зміни вносились на основі стану з попередньої ітерації. Для кожного нетермінала перебираються його виведення, і для кожного виведення обчислюються можливі рядки довжини до k. Ці рядки додаються до множини First_k відповідного нетермінала. Ітерації циклу продовжуються, поки множини First_k не перестануть змінюватися між ітераціями:

```
def compute_first_k(self, k):
    first = {}
    prev_first = {}
    # Initialize the first set for each non-terminal in the grammar to be empty.
    for n in self.grammar.non_terminals:
        first[n] = set()
    # Loop until the first set no longer changes between iterations.
    while first != prev_first:
        prev_first = first.copy()
        # Create a value copy of each set in the dictionary to avoid modifying the original sets during iteration.
        for key, value in prev_first.items():
            prev_first[key] = value.copy()
        for n in self.grammar.non_terminals:
            for production in n.productions:
                # Calculate possible strings of length k or less from the production using the current first set.
                possible_strings = self.get_possible_strings(production, k, first)
                # Update the first set of the non-terminal by adding the new possible strings.
                first[n] |= set(possible_strings)

    return first
```

Функція compute_follow_k реалізує алгоритм обчислення множини Follow_k для кожного нетермінала у граматиці. Множина Follow_k включає всі можливі рядки довжини до k, які можуть виникати відразу після даного нетермінала в деяких виведеннях граматики. Для стартового символу до його множини follow додається символ кінця рядка. Функція виконує ітераційний процес, де на кожній ітерації обчислюється Follow_k для кожного нетермінала на основі його виведень та вже визначених Follow_k для інших нетерміналів. Перед кожною ітерацією створюється копія поточного стану

множини follow, щоб зміни вносились на основі стану з попередньої ітерації. Для кожного нетермінала перебираються його виведення, і для кожного символа в виведенні обчислюються можливі рядки, які можуть йти після нього, довжиною до k. Ці рядки додаються до множини Follow_k відповідного нетермінала. Ітерації циклу продовжуються до тих пір, поки множини Follow_k не перестануть змінюватися між ітераціями:

```
def compute_follow_k(self, k, first_k):
    follow = {}
    prev_follow = {}
    # Initialize the follow set for each non-terminal in the grammar.
    for n in self.grammar.non_terminals:
        follow[n] = set()
    # For the start symbol, add epsilon to its follow set.
    follow[self.grammar.non_terminals[0]].add((self.grammar.get_epsilon(),))

    # Initialize a set to track which non-terminals have been seen.
    seen_non_terminals = set()
    # Add the start symbol to the seen non-terminals set.
    seen_non_terminals.add(self.grammar.non_terminals[0])

    # Loop until the follow set no longer changes between iterations.
    while follow != prev_follow:
        prev_follow = follow.copy()
        for key, value in prev_follow.items():
            # Create a value copy of each set in the dictionary to
            # avoid modifying the original sets during iteration.
            prev_follow[key] = value.copy()

        # Store newly seen non-terminals during this iteration.
        new_seen_non_terminals = []
        for nt in seen_non_terminals:
            for production in nt.productions:
                for i, c in enumerate(production):
                    # Only process non-terminal symbols.
                    if isinstance(c, NonTerminal):
                        # Add the non-terminal to the buffer of newly seen non-terminals.
                        new_seen_non_terminals.append(c)
                        # Get the symbols following the current non-terminal in the production.
                        after = production[i + 1:]
                        # Calculate the first set of the string following the current non-terminal.
                        first_of_after = self.concatenate_k(k, [ps[:k] for ps in
                                                                self.get_possible_strings(after, k, first_k)]
                                                            follow[nt])
                        # Update the follow set of the current non-terminal.
                        for s in first_of_after:
                            if len(s) == 0:
                                # If the string is empty, add epsilon.
                                follow[c].add((self.grammar.get_epsilon(),))
                            else:
                                follow[c].add(s)

        seen_non_terminals |= set(new_seen_non_terminals)
```

Функція concatenate_k використовується для об'єднання двох множин рядків у контексті обчислення множин Follow_k та First_k. Для кожної пари рядків s1 з першої множини та s2 з другої множини виконується об'єднання. Якщо обидва рядки є пустими (тобто містять лише епсилон), то в результат додається епсилон. Якщо тільки один з рядків є пустим, то до результату додається непустий рядок. Якщо жоден з рядків не є пустим, вони конкатенуються, і до результату додається урізаний рядок з довжиною не більше k:

```
def concatenate_k(self, k, first_set, second_set):
    result = set()
    for s1 in first_set:
        for s2 in second_set:
            # Check if all symbols in s1 and s2 are empty (epsilon).
            s1_empty = all(c.is_empty() for c in s1)
            s2_empty = all(c.is_empty() for c in s2)
            # If both strings are empty, add an epsilon to the result set.
            if s1_empty and s2_empty:
                result.add((self.grammar.get_epsilon(),))
                continue
            # If only the first string is empty, add the second string to the result set.
            elif s1_empty:
                result.add(s2)
                continue
            # If only the second string is empty, add the first string to the result set.
            elif s2_empty:
                result.add(s1)
                continue
            # Concatenate the two strings and add the result to the result set.
            result.add((s1 + s2)[:k])
    return result
```

Функція get_possible_strings використовується для генерування всіх можливих рядків, що випливають з заданого виведення в граматиці з обмеженням довжини до k символів. Створюється порожній список possible_strings для зберігання можливих рядків та черга queue для обробки виведень. В чергу додається перший варіант виведення для обробки. Поки черга не пуста, з неї вилучається поточне виведення для обробки. Якщо перші k символів поточного виведення є терміналами або порожніми, вони додаються до possible_strings як можливий рядок. Для кожного нетермінального символу в виведенні створюється новий варіант виведення, замінюючи нетермінал його можливими початковими рядками. Кожен новий варіант виведення додається до черги для подальшої обробки. Після обробки

всіх елементів черги функція повертає список possible_strings з усіма можливими рядками:

```
    def get_possible_strings(self, production, k, prev_first_k):
        possible_strings = []
        queue = deque()
        # Start with the original production.
        queue.append(list(production))
        while queue:
            # Take the first item from the queue for processing.
            current_production = queue.popleft()
            # Check if the first k symbols are all terminals or empty.
            if all(isinstance(c, Terminal) for c in current_production[:k]):
                if all(nt_c.is_empty() for nt_c in current_production[:k]):
                    # If all symbols are empty, add epsilon to possible strings.
                    possible_strings.append((self.grammar.get_epsilon(),))
                else:
                    # Add the first k terminals as a possible string.
                    possible_strings.append(tuple(current_production[:k]))
                continue
            # Iterate through each symbol in the current production.
            for i, c in enumerate(current_production):
                # Process only non-terminal symbols.
                if isinstance(c, NonTerminal):
                    # For each possible first set of the non-terminal, create a new production variant.
                    for nt_first in prev_first_k[c]:
                        new_production = current_production.copy()
                    # Check if all symbols in the first set of non-terminal are empty.
                    is_prev_first_empty = all(nt_c.is_empty() for nt_c in nt_first)
                    # Handle empty symbols in non-terminal expansions.
                    if is_prev_first_empty and len(current_production) > 1:
                        # If the first set is empty and production is not a single symbol, remove the non-terminal.
                        new_production[i:i + 1] = []
                    else:
                        # Replace the non-terminal with its first set.
                        new_production[i:i + 1] = nt_first
                    # Add the new production variant to the queue for further processing.
                    queue.append(new_production)
            break
return possible_strings
```

Допоміжна функція, яка перетворює кортежі в рядки:

```
@staticmethod
def tuples_to_strings(table):
    result = {}
    for key, value in table.items():
        result[key] = set()
        for v in value:
            result[key].add(''.join([str(c) for c in v]))
    return result
```

Наступним буде створення класу ConstructParsingTable для побудови таблиці синтаксичного аналізатору. Ось його ініціалізація:

```python
class ConstructParsingTable:
    oyl04
    def __init__(self, grammar, first_sets, follow_sets):
        self.grammar = grammar
        self.first_sets = first_sets
        self.follow_sets = follow_sets
        self.parsing_table = {}
```

Функція construct використовується для побудови таблиці синтаксичного аналізу для LL(1) парсера, яка визначає, яке виведення слід використовувати при аналізі певного нетермінала, залежно від поточного стеку.

Функція ітерується через всі нетермінали та їх виведення у граматиці. Для кожного виведення визначається множина First_k. Виведення додаються до таблиці синтаксичного аналізу на основі множини First_k, якщо вони не ведуть до епсилона. Якщо множина First_k містить епсилон, то виконується додаткова обробка для відповідних виведень.

```python
def construct(self):
    # Construct the parsing table for each non-terminal and its productions.
    for non_terminal, productions in self.grammar.productions.items():
        for production in productions:
            # Find the first set for the production.
            first_production = self._find_first_production(production)
            # Add productions to the parsing table based on the first set.
            for terminal_tuple in first_production:
                terminal = terminal_tuple[0]
                if terminal != Terminal('ε'):
                    self._add_to_parsing_table(non_terminal, terminal, production)
            # Process productions that can derive epsilon.
            if (Terminal('ε'),) in first_production:
                self._process_epsilon(non_terminal, production)

    return self.parsing_table
```

Функція _find_first_production визначає множину First_k для конкретного виведення в граматиці. Множина First_k складається з терміналів, які можуть з'явитися на початку рядків, що виводяться з цього виведення. Алгоритм:

Виведення перетворюється у список символів для подальшої обробки. Для кожного символу в виведенні визначається, чи є він терміналом чи нетерміналом. Для кожного символу визначається множина First_k. Множина first_production оновлюється за допомогою множини First_k кожного

символу, поки не зустріне символ, що не може вивести епсилон. Якщо всі символи в виведенні можуть вивести епсилон, до множини додається епсилон:

```
def _find_first_production(self, production):
    first_production = set()
    # Convert the production into a list of symbol texts.
    production = [c.text for c in Grammar.get_tnt_string(self.grammar, production)]

    for t_symbol in production:
        # Determine if the symbol is a terminal or non-terminal.
        symbol = Terminal(t_symbol) if Terminal(t_symbol) in self.grammar.terminals else NonTerminal(t_symbol)
        # Get the first set of the symbol.
        first_symbol = self.first_sets[symbol] if symbol in self.first_sets else {(symbol,)}

        if (Terminal('ε'),) not in first_symbol:
            first_production.update(first_symbol)
        # Stop if the symbol cannot derive epsilon.
        if (Terminal('ε'),) not in first_symbol:
            break
    else:
        # Add epsilon if all symbols in the production can derive epsilon.
        first_production.add((Terminal('ε'),))
    return first_production
```

Функція _add_to_parsing_table відповідає за додавання виведення до таблиці синтаксичного аналізу за певним нетерміналом і терміналом. Перш ніж додати виведення, функція перевіряє, чи не існує вже запису для цієї пари нетермінал-термінал, щоб уникнути конфліктів. Якщо конфліктів немає, виведення додається до таблиці.  Якщо виникає конфлікт, генерується помилка, яка сигналізує про те що граматика не є LL(1).

Функція _process_epsilon відповідає за обробку виведень, що виводять епсилон, та додавання відповідних записів до таблиці аналізу. Для нетермінала використовується множина Follow_k для додавання виведень, які можуть вивести епсилон. Виведення, що виводять епсилон, додаються до таблиці для відповідних терміналів з множини Follow_k. Якщо в таблиці вже існує запис для даної пари нетермінал-термінал, генерується помилка про те, що граматика не є LL(1).

```python
def _add_to_parsing_table(self, non_terminal, terminal, production):
    # Add a production to the parsing table.
    production = [c.text for c in Grammar.get_tnt_string(self.grammar, production)]
    # Check for conflicts in the parsing table.
    if (non_terminal, terminal) not in self.parsing_table:
        self.parsing_table[(non_terminal, terminal)] = production
    else:
        raise ValueError(f"Grammar is not LL(1): Conflict at ({non_terminal}, {terminal})")

1 usage  oyl04
def _process_epsilon(self, non_terminal, production):
    # Process productions that can derive epsilon.
    production = [c.text for c in Grammar.get_tnt_string(self.grammar, production)]
    # Use the follow set of the non-terminal to add productions for deriving epsilon.
    for terminal in self.follow_sets[NonTerminal(non_terminal)]:
        if (non_terminal, terminal[0]) not in self.parsing_table:
            self.parsing_table[(non_terminal, terminal[0])] = production
        else:
            raise ValueError(f"Grammar is not LL(1): Conflict at ({non_terminal}, {terminal})")
```

Наступною буде функція для аналізу виразів за допомогою контрольної таблиці, яка керує процесом парсингу. Вона використовує клас FirstFollow для обчислення перших та наступних наборів для граматики, які є необхідними для побудови таблиці парсингу для LL(k) парсерів. За допомогою граматики та обчислених First_k та Follow_k множин створюється контрольна таблиця. Функція намагається створити LLkAnalyzer, який використовує контрольну таблицю та граматику для парсингу заданого виразу. Вираз токенізується та передається аналізатору. Якщо вираз відповідає правилам граматики, він буде успішно проаналізований, і виведення, застосовані під час процесу парсингу, повертаються. Інакше виникне помилка.

```python
def parse_with_control_table(grammar, expression):
    first_follow = FirstFollow(grammar)
    first_k = first_follow.compute_first_k(1)
    follow_k = first_follow.compute_follow_k( k: 1, first_k)
    parser = ConstructParsingTable(grammar, first_k, follow_k)
    table = parser.construct()
    print("Parser control table:")
    for key, value in table.items():
        print(str(key) + ":")
        joined_value = ''.join(map(str, value))
        print(f"{str(key[0])} -> {'ε' if not joined_value else joined_value}")
        print("")
    recursive_parser = RecursiveDescentParser(grammar)
    print(f'Recursive Descent Parsing "{expression}": {recursive_parser.parse(expression)}')
    print("Analyzer process:")
    applied_productions = None
    try:
        analyzer = LLkAnalyzer(table, grammar)
        tokenized = grammar.get_tnt_string(expression)
        applied_productions = analyzer.parse(tokenized)
    except SyntaxError as se:
        print(f"Got an error: {str(se)}")
    except ValueError as ve:
        print(f"Got an error: {str(ve)}")
    return applied_productions
```

Створимо клас для LL(k)-аналізатора, напишемо для нього методи ініціалізації з даною граматикою та таблицею парсера, а також встановимо початковий нетермінал:

```
class LLkAnalyzer:
    def __init__(self, parsing_table, grammar):
        self.parsing_table = parsing_table
        self.grammar = grammar
        self.start_symbol = self.grammar.non_terminals[0]

    def str_to_sym(self, sym):
        if Terminal(sym) in self.grammar.terminals:
            return Terminal(sym)
        else:
            return NonTerminal(sym)
```

Метод парсингу буде доволі великим, тому розглянемо його частинами. В першій частині ініціалізуються змінні, до токенів додається символ закінчення, стек ініціалізується символом закінчення та стартовим символом. Цикл працює, поки стек не стане порожнім. На кожному кроці буде виводитися номер поточного кроку, вміст стека і токени, які ще не були проаналізовані. Якщо верхній елемент стека є термінальним символом, то парсер намагається зіставити його з поточним токеном. Якщо термінальний символ відповідає поточному токену, то переходимо до наступного індексу. Якщо зіставлення не сталося, то генерується помилка “SyntaxError” з повідпомленням про неочікуваний токен, тобто обраний вираз не можна вивести в граматиці з файлу.

```
def parse(self, tokens):
    tokens.append('$')  # Append end marker to the token list
    applied_productions = []
    stack = ['$', self.start_symbol]  # Initialize stack with end marker and start symbol

    current_token_index = 0
    step = 0
    while len(stack) > 0:
        print(f"Step #{step}")
        step += 1
        print(list(reversed(stack)))
        print(tokens[current_token_index:])
        top = stack.pop()
        # Prints for current stack, top element, and remaining tokens

        if isinstance(top, Terminal):
            #  Match terminal symbol with current token
            if top == tokens[current_token_index]:
                current_token_index += 1
            else:
                raise SyntaxError(f"Unexpected token: Expected {top}, found {tokens[current_token_index]}")
```

Наступна частина обробляє нетермінальні символи в аналізаторі. Коли верхній елемент стека є нетермінальним, метод спочатку перевіряє, чи поточний токен є кінцевим маркером '$'. Якщо так, то метод шукає у таблиці аналізу запис, що відповідає нетерміналу та епсилону. Якщо запис знайдено, це означає, що нетермінал можна замінити на епсілон, тобто ігнорувати його в подальшому аналізі. Якщо запис у таблиці аналізу знайдено, метод додає це виведення до списку applied_productions. Він проходить по символах у знайденому виведенні в зворотньому порядку, ігноруючи епсилон, і додає кожен символ до стека, конвертуючи їх за допомогою методу str_to_sym.

Далі, якщо запис у таблиці аналізу знайдено, це означає, що існує виведення, яке можна застосувати для поточного нетерміналу та поточного вхідного токена. Метод знову додає це виведення до списку applied_productions і проходить по символах у знайденому виведенні в зворотньому порядку, ігноруючи епсілони, і додає кожен символ до стека, конвертуючи їх за допомогою методу str_to_sym. Ця частина методу відповідає за знаходження відповідностей між нетерміналами в стеку та правилами в граматиці, використовуючи таблицю аналізу для керування процесом аналізу.

```
elif isinstance(top, NonTerminal):
    #  Process non-terminal
    #  Special processing when the current token is the end marker '$'
    if isinstance(tokens[current_token_index], str) and tokens[current_token_index] == '$':
        #  Look up in the parsing table for an entry with the non-terminal and epsilon
        entry = self.parsing_table.get((str(top), Terminal('ε')))
        #  If an entry is found, it means the non-terminal can be replaced with epsilon
        if entry is not None:
            applied_productions.append((str(top), Terminal('ε'), entry))
            for symbol in reversed(entry):
                if symbol == 'ε':
                    continue
                stack.append(self.str_to_sym(symbol))
            continue
    # If an entry is found in the parsing table, it means there is a production that can be applied
    # for the current non-terminal (top of the stack) and the current input token.
    entry = self.parsing_table.get((str(top), tokens[current_token_index]))
    if entry is not None:
        applied_productions.append((str(top), tokens[current_token_index], entry))
        # Iterate through the symbols in the found production in reverse order.
        for symbol in reversed(entry):
            # If the symbol is epsilon -> we do not need to match any input token,
            # so we can skip it.
            if symbol == 'ε':
                continue
            # For other symbols, convert them to terminals or non-terminals and push them onto the stack.
            stack.append(self.str_to_sym(symbol))
```

Якщо під час обробки нетерміналу у таблиці аналізу для поточного верхнього елемента стека не знаходиться відповідного запису, потрібно

вивести помилку SyntaxError, що сигналізує про відсутність коректного виведення для обробки цього нетерміналу з поточним токеном. Коли верхній елемент стека є кінцевим маркером '$', і поточний токен також є кінцевим маркером, то аналіз вважається успішно завершеним. Якщо поточний токен не є кінцевим маркером, то потрібно вивести помилку SyntaxError з повідомленням про несподіваний кінець вводу. Якщо ніяка з умов не спрацювала, то буде помилка ValueError, яка сигналізує про недійсний символ на стеку. Наступною йде перевірка, чи вхідний рядок був повністю проаналізований, якщо ні, то потрібно вивести помилку SyntaxError. В кінці створюється список terms, в якому кожне виведення конвертується в термінали та нетермінали і додається до списку terms. Ця частина відповідає за заключну частину аналізу, повноту проаналізованих даних та підготовці даних для повернення

```
            else:
                # If no entry is found in the parsing table, it means there is no valid production for the current
                raise SyntaxError(f"No production to parse: {top} with token {tokens[current_token_index]}")
        # This condition checks if we have reached the end of the stack.
        elif top == '$':
            if tokens[current_token_index] == '$':
                print("Parsing successful!")
            # If the current token is also the end marker '$', it means the input string
            # has been successfully parsed according to the grammar productions.
            else:
                raise SyntaxError("Unexpected end of input")

        else:
            raise ValueError(f"Invalid symbol on stack: {top}")

    if current_token_index < len(tokens) - 1:
        raise SyntaxError("Input not fully parsed")

    terms = []
    #  Convert applied productions to terminals and non-terminals
    for production in applied_productions:
        func = lambda x: list(map(self.str_to_sym, x))
        terms.append((func(production[0]), production[1], func(production[2])))
    return terms
```

Створимо клас для аналізатора методом рекурсивного спуску.

Першим методом буде конструктор, який ініціалізує парсер з граматикою, наданою йому і створює список для токенів.

Наступний метод parse, він буде розбирати вираз згідно з граматикою, він розбиває вхідний рядок на токени, визначає стартовий символ граматики і намагається розібрати весь список токенів від цього стартового символу.

Метод parse_non_terminal використовується для розбору нетермінального символу, він зберігає поточний індекс, ітерується через виведення всіх нетерміналів.

parse_symbol викликається для розбору символу, який може бути і термінальним, і нетермінальним. В залежності від терміналу використовує відповідний метод для його розбору

match використовується для співставлення термінального символу з поточним токеном. Він перевіряє чи поточний токен відповідає терміналу, і якщо це так, то переходить до наступного індексу.

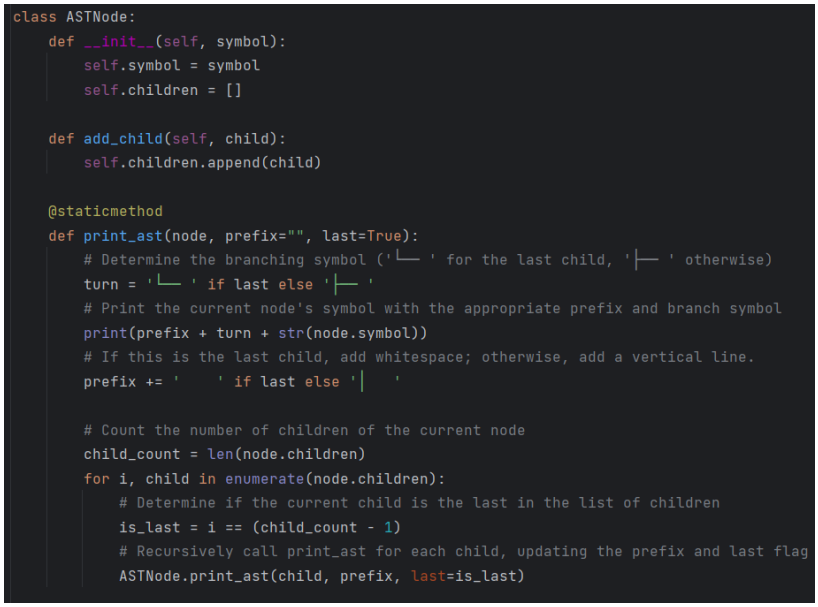
```
class RecursiveDescentParser:
    def __init__(self, grammar):
        self.grammar = grammar
        self.tokens = []
        self.index = 0

    def parse(self, input_str):
        # Convert input string into tokens based on the grammar
        self.tokens = [c.text for c in self.grammar.get_tnt_string(input_str)]
        self.index = 0
        # Start symbol is the first non-terminal in grammar
        start_symbol = self.grammar.non_terminals[0]
        # Check if the entire token list can be parsed from the start symbol
        return self.parse_non_terminal(start_symbol) and self.index == len(self.tokens)

    def parse_non_terminal(self, non_terminal):
        save_index = self.index
        # Try each production of the non-terminal to see if it matches the tokens
        for production in non_terminal.productions:
            self.index = save_index  # Reset index to try next production
            if all(self.parse_symbol(symbol) for symbol in production):
                return True  # Successful parsing of this non-terminal
        self.index = save_index
        return False  # This non-terminal can't be parsed

    def parse_symbol(self, symbol):
        if isinstance(symbol, Terminal):
            if symbol.is_empty():
                return True
            # Match terminal with current token
            return self.match(symbol.text)
        elif isinstance(symbol, NonTerminal):
            # Recursive parsing for non-terminal
            return self.parse_non_terminal(symbol)
        # If symbol type is unrecognized, return False
        return False

    def match(self, terminal):
        # Check if the current token matches the terminal
        if self.index < len(self.tokens) and self.tokens[self.index] == terminal:
            self.index += 1
            return True  # Successful match
        return False
```

Створимо клас вузла абстрактного синтаксичного дерева (ASTNode). Конструктор ініціалізує об’єкт з заданим символом і порожнім списком дочірніх вузлів.

Метод add_child додає дочірній вузол до списку.

Метод print_ast відображає AST з даного вузла у структурованому виді, який показує ієрархію вузлів. └── використовується для останніх дочірніх

вузлів, а ├── у інших випадках. Функція виводить символ поточного вузла з символом розгалуження, а потім рекурсивно викликається для кожного дочірнього вузла:



```python
class ASTNode:
    def __init__(self, symbol):
        self.symbol = symbol
        self.children = []

    def add_child(self, child):
        self.children.append(child)

    @staticmethod
    def print_ast(node, prefix="", last=True):
        # Determine the branching symbol ('└── ' for the last child, '├── ' otherwise)
        turn = '└── ' if last else '├── '
        # Print the current node's symbol with the appropriate prefix and branch symbol
        print(prefix + turn + str(node.symbol))
        # If this is the last child, add whitespace; otherwise, add a vertical line.
        prefix += '    ' if last else '│   '

        # Count the number of children of the current node
        child_count = len(node.children)
        for i, child in enumerate(node.children):
            # Determine if the current child is the last in the list of children
            is_last = i == (child_count - 1)
            # Recursively call print_ast for each child, updating the prefix and last flag
            ASTNode.print_ast(child, prefix, last=is_last)
```

Також потрібно додати метод build_ast в клас граматики.

Створюється вузол кореня дерева з першого нетермінала граматики і поміщається в стек. Далі з кожного правила в послідовності правил отримуємо нетермінал, токен і виведення. Поточний вузол отримується зі стеку, і якщо символ вузла відповідає нетерміналу правила, обробка продовжується. Для кожного символу в виведенні створюється новий вузол AST і додається як дочірній вузол до поточного вузла. Це робиться в зворотному порядку, оскільки дочірні вузли повинні бути оброблені зліва направо, а стек працює за принципом LIFO (останній прийшов - перший пішов). Якщо дитина є нетермінальним вузлом, вона знову додається до стеку для подальшої обробки. Після завершення обходу всіх правил функція повертає вузол кореня, який представляє побудоване AST:

```
1 usage (1 dynamic)  oyl04
def build_ast(self, rule_sequence):
    root = ASTNode(self.non_terminals[0])
    stack = [root]
    for rule in rule_sequence:
        # Unpack the rule into non-terminal, token, and production.
        nt, _, prod = rule
        nt = nt[0]
        curr_node = stack.pop()
        if curr_node.symbol == nt:
            # Iterate over the production in reverse order.
            # This is because we want to process children from left to right, and the stack is LIFO.
            for c in prod[::-1]:
                # For each symbol in the production, create a new AST node and add it as a child.
                if isinstance(c, Terminal):
                    curr_node.add_child(ASTNode(c))
                else:
                    curr_node.add_child(ASTNode(c))
                    # If the child is a non-terminal, push it onto the stack for further processing.
                    stack.append(curr_node.children[-1])
    return root
```

Останньою створеною функцією буде функція, яка буде друкувати побудовані дані:

```
def view_result(grammar, expression):
    first_follow = FirstFollow(grammar)
    first_k = first_follow.compute_first_k(1)
    follow_k = first_follow.compute_follow_k( k: 1, first_k)
    print("Terminals:")
    print(grammar.terminals)
    print("Non-Terminals:")
    print(grammar.non_terminals)
    print("Epsilon-Producers:")
    print(grammar.nullable_non_terminals)
    print("First(k):")
    for n in grammar.non_terminals:
        print(str(n) + ":")
        print(', '.join(''.join(map(str, sym)) for sym in first_k[n]))
    print("Follow(k):")
    for n in grammar.non_terminals:
        print(str(n) + ":")
        print(', '.join(''.join(map(str, sym)) for sym in follow_k[n]))
    order_of_rules = parse_with_control_table(grammar, expression)
    if order_of_rules:
        print("Applied Rules:")
        for i, rule in enumerate(order_of_rules):
            print(f"{rule[0]} -> {rule[2]}")
        print("Abstract Syntax Tree:")
        root = grammar.build_ast(order_of_rules)
        ASTNode.print_ast(root)
    else:
        print("Impossible to build AST")
```

Щоб запустити програму відкриємо файл з граматикою, оберемо вираз, який потрібно вивести і запустимо функцію виводу результатів, яка потім буде викликати інші функції програми:

```
if __name__ == "__main__":
    example_grammar = Grammar(Grammar.read_grammar_from_file("in.txt"))
    example_expression = "(a+a)*a"
    view_result(example_grammar, example_expression)
```

Візьмемо для прикладу таку граматику:

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

І потрібно вивести слово (a+a)*a

Програма виводить такі термінали, нетермінали та епсилон-нетермінали

```
[+, ), ε, (, a, *] [S, A, B, C, D]
Terminals:
[+, ), ε, (, a, *]
Non-Terminals:
[S, A, B, C, D]
Epsilon-Producers:
{C, A}
```

Далі вона виводить множини First(k) і Follow(k) для k = 1.

Наступним кроком виведення є побудова таблиці контролю:

```
First(k):  Follow(k):
S:         S:
a, (       ), ε
A:         A:
+, ε       ), ε
B:         B:
a, (       ), +, ε
C:         C:
ε, *       ), +, ε
D:         D:
a, (       ε, ), +, *
```

```
Parser control table:
('S', a):
S -> BA

('S', (): ('B', a):  ('C', +):
S -> BA   B -> DC    C -> ε

('A', +): ('B', ():  ('C', ε):
A -> +BA  B -> DC    C -> ε

('A', )): ('C', *):  ('D', ():
A -> ε    C -> *DC   D -> (S)

('A', ε): ('C', )):  ('D', a):
A -> ε    C -> ε     D -> a
```

За допомогою рекурсивного спуску перевірити чи можна вивести це слово у граматиці. Далі будуть кроки аналізатора по таблиці:

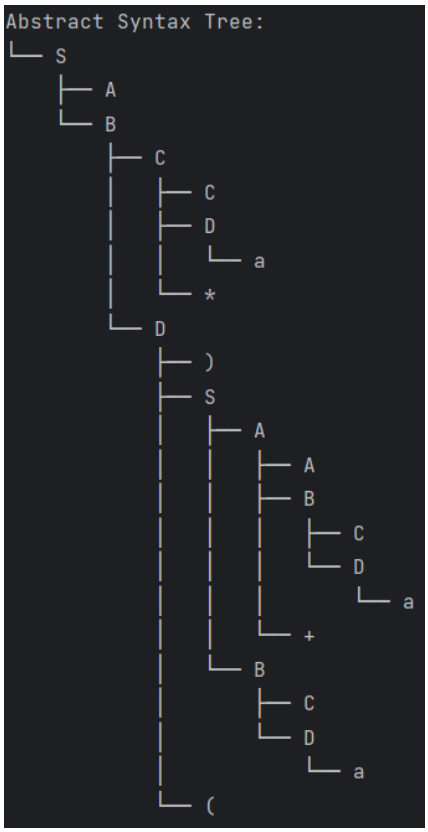
```
Recursive Descent Parsing "(a+a)*a": True
Analyzer process:
```

```
Step #0
[S, '$']
[(, a, +, a, ), *, a, '$']
Step #1
[B, A, '$']
[(, a, +, a, ), *, a, '$']
Step #2
[D, C, A, '$']
[(, a, +, a, ), *, a, '$']
Step #3
[(, S, ), C, A, '$']
[(, a, +, a, ), *, a, '$']
Step #4
[S, ), C, A, '$']
[a, +, a, ), *, a, '$']
Step #5
[B, A, ), C, A, '$']
[a, +, a, ), *, a, '$']
Step #6
[D, C, A, ), C, A, '$']
[a, +, a, ), *, a, '$']
Step #7
[a, C, A, ), C, A, '$']
[a, +, a, ), *, a, '$']
Step #8
[C, A, ), C, A, '$']
[+, a, ), *, a, '$']
Step #9
[A, ), C, A, '$']
[+, a, ), *, a, '$']
Step #10
[+, B, A, ), C, A, '$']
[+, a, ), *, a, '$']
Step #11
[B, A, ), C, A, '$']
[a, ), *, a, '$']
Step #12
[D, C, A, ), C, A, '$']
[a, ), *, a, '$']
Step #13
[a, C, A, ), C, A, '$']
[a, ), *, a, '$']
Step #14
[C, A, ), C, A, '$']
[), *, a, '$']
Step #15
[A, ), C, A, '$']
[), *, a, '$']
Step #16
[), C, A, '$']
[), *, a, '$']
Step #17
[C, A, '$']
[*, a, '$']
Step #18
[*, D, C, A, '$']
[*, a, '$']
Step #19
[D, C, A, '$']
[a, '$']
Step #20
[a, C, A, '$']
[a, '$']
Step #21
[C, A, '$']
['$']
Step #22
[A, '$']
['$']
Step #23
['$']
['$']
```

Послідовність правил для виведення (a+a)*a:

```
Parsing successful!
Applied Rules:
[S] -> [B, A]
[B] -> [D, C]
[D] -> [(, S, )]
[S] -> [B, A]
[B] -> [D, C]
[D] -> [a]
[C] -> []
[A] -> [+, B, A]
[B] -> [D, C]
[D] -> [a]
[C] -> []
[A] -> []
[C] -> [*, D, C]
[D] -> [a]
[C] -> []
[A] -> []
```

Абстрактне синтаксичне дерево для заданої граматики:




## ВИСНОВОК

У ході виконання лабораторної роботи №4 з дисципліни "Системне програмування" було розроблено LL(1)-синтаксичний аналізатор. Метою роботи було створення інструменту, здатного аналізувати вхідні рядки на

відповідність заданій граматиці та будувати абстрактне синтаксичне дерево (AST) або визначати синтаксичні помилки.

Під час виконання лабораторної було досягнуто наступного:

Реалізовано основні функції First(k) та Follow(k) для обчислення множин, що дозволяють ефективно управляти процесом синтаксичного аналізу.

Запрограмовано допоміжні функції для роботи з граматикою та її розбору, що забезпечує гнучкість та розширюваність аналізатора.

Розроблено механізм рекурсивного спуску, який дозволяє аналізатору визначати коректність виразів, базуючись на заданій граматиці.

Реалізовано візуалізацію абстрактного синтаксичного дерева, що надає зрозуміле графічне представлення структури вхідного коду.

Створений аналізатор є ефективним інструментом для аналізу граматик та може бути використаний як основа для подальших досліджень у галузі системного програмування.