

Матеріали до виконання  
лабораторної роботи № 3  
Синтаксичний та семантичний аналіз програм  
*Частина 1. Синтаксичний аналіз методом  
рекурсивного спуску (№ 3.1)*

Юрій Стативка

Вересень, 2024 р.

## Зміст

<b>Вступ</b>	<b>1</b>
<b>1 Початкові дані для розробки парсера</b>	<b>2</b>
1.1 Граматика мови . . . . .	3
1.2 Лексичний аналізатор мови . . . . .	3
1.3 Формат таблиці символів програми . . . . .	4
1.4 Приклади програм для тестування . . . . .	5
1.5 Код парсера . . . . .	5
<b>2 Програмна реалізація парсера</b>	<b>5</b>
2.1 Перші кроки . . . . .	5
2.2 Заглушка для <code>parseStatementList()</code> . . . . .	8
2.3 Вниз від <code>parseStatementList()</code> . . . . .	13
2.4 <code>Expression</code> без лівої рекурсії . . . . .	15
2.5 Отже . . . . .	18
<b>3 Оформлення та зміст звіту</b>	<b>20</b>
<b>Література</b>	<b>21</b>

## Вступ

У цьому тексті розглядаються тільки питання побудови **синтаксичного аналізатора методом рекурсивного спуску**. Код двох розглянутих прикладів парсера, — з заглушкою та без заглушки, додаються у архіві.

**Побудова семантичного аналізатора**, яка складає другу частину (№ 3.2) цієї роботи, буде описана у іншому документі.

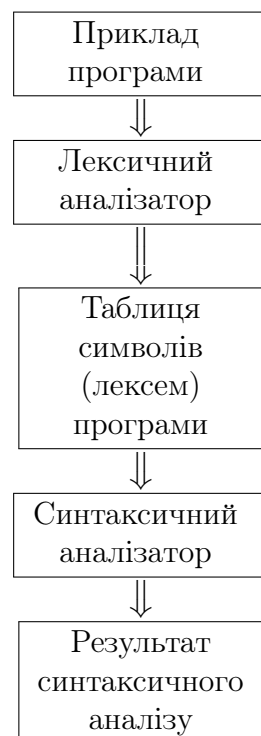
Ті, хто вже зрозумів як виконати це завдання, чи вже виконав його, можуть відразу перейти до перегляду вимог до оформлення звіту у розділі 3.

**Мета роботи** – програмна реалізація синтаксичного аналізатора (парсера) заданої мови методом рекурсивного спуску.

Для виконання роботи потрібні:

1. Специфікація мови.
2. Лексичний аналізатор мови (ЛР2).
3. Формат таблиці символів (таблиці лексем, таблиця розбору), яку повертає лексичний аналізатор (див. п.2).
4. Приклади програм для тестування синтаксичного аналізатора.

Входом синтаксичного аналізатора є таблиця символів програми. Результатом синтаксичного аналізу, у цій роботі, — відповідь про синтаксичну коректність чи некоректність програми та повідомлення про виявлені синтаксичні помилки.



## 1 Початкові дані для розробки парсера

У цьому тексті визначається мова `MicroLang2`, яка відрізняється від `MicroLang1`, про яку йшлося у [1]. Зокрема, тут додано інструкцію розгалуження `IfStatement`<sup>1</sup>, логічний вираз `BoolExpr` та оператори відношення `RelOp`.

<sup>1</sup>Зверніть увагу, що після `then` та після `else` обов'язкова єдина інструкція `Statement`, а не `StatementList`, як можна було би сподіватись.

## 1.1 Граматика мови

```
Program = program StatementList end

StatementList = Statement { Statement }

Statement = Assign
           | IfStatement

Assign = Ident ':' Expression

Ident = Letter {Letter | Digit }

Expression = Expression ('+' | '-' | '*' | '/') Expression
           | Id
           | Const
           | '(' Expression ')'

IfStatement = if BoolExpr then Statement else Statement endif

BoolExpr = Expression RelOp Expression

RelOp = '=' | '<=' | '>=' | '<' | '>' | '<>'

Const = Float | Int

Float = Digit {Digit} '.' {Digit}

Int = Digit {Digit}

Letter = a | b | c | d | e | f | g | h | i | j | k | l | m
       | n | o | p | q | r | s | t | u | v | w | x | y | z

Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

WhiteSpace = (Blank|Tab) {Blank|Tab}

Blank = " "

Tab = '\t'

Newline = '\n'
```

## 1.2 Лексичний аналізатор мови

Використовується приклад лексичного аналізатора до роботи комп'ютерного практикуму № 2, реалізований мовою python, див. [1], розширений для обробки

інструкцій розгалуження `IfStatement` та логічних виразів `BoolExpr`. Лексичний аналізатор (сканер, лексер) та таблиця символів програми імпортуються так:

```
from lex_my_lang import lex
from lex_my_lang import tableOfSymb
```

Виклик сканера

```
FSucces = lex()
```

приводить до присвоєння ознаці успішності лексичного розбору `FSucces` значення ('Lexer', True), або, у випадку виявлення лексичної помилки, значення ('Lexer', False).

### 1.3 Формат таблиці символів програми

Таблиця символів програми `tableOfSymb` – це словник (dictionary) мови python. Формат був визначений при розробці лексичного аналізатора:

```
{ n_rec : (num_line, lexeme, token, idxIdConst) }
```

де:

`n_rec` – номер запису в таблиці символів програми;

`num_line` – номер рядка програми;

`lexeme` – лексема;

`token` – токен лексеми;

`idxIdConst` – індекс ідентифікатора або константи у таблиці ідентифікаторів та констант відповідно.

Наприклад, для програми з п'яти рядків:

```
1 program
2
3 v1:= -(5.4 + 3)
4
5 end
```

лексичний аналізатор будує таблицю символів, яка синтаксичним аналізатором виводиться у консоль командою

```
print('tableOfSymb:0'.format(tableOfSymb)):
```

```
{1: (1, 'program', 'keyword', ''), 2: (3, 'v1', 'id', 1),
 3: (3, ':=', 'assign_op', ''), 4: (3, '-', 'add_op', ''),
 5: (3, '(', 'par_op', ''), 6: (3, '5.4', 'float', 1),
 7: (3, '+', 'add_op', ''),
 8: (3, '3', 'int', 2), 9: (3, ')', 'par_op', ''),
10: (5, 'end', 'keyword', '')}.
```

Більш зручний для читання варіант виводиться лексичним аналізатором у форматі

`num_line lexeme token idxIdConst` (без номера запису у тій таблиці розбору `n_rec`):

```

1  program      keyword
3  v1           id           1
3  :=          assign_op
3  -           add_op
3  (           par_op
3  5.4         float        1
3  +           add_op
3  3           int          2
3  )           par_op
5  end         keyword

```

## 1.4 Приклади програм для тестування

Тестування парсера передбачає синтаксичний розбір прикладів коду та оцінку його синтаксичної правильності (висновок про синтаксичну коректність чи некоректність коду, повідомлення про знайдену помилку, її локалізацію, діагностичне повідомлення тощо).

## 1.5 Код парсера

Всі наведені далі приклади можна знайти у теці `my_lang_parser`. Перший, з функцією-заглушкою, – у `my_lang_parser\parser_01.StatementList=stub`. Другий, з повною реалізацією парсера для наведеної граматики, – у теці `my_lang_parser\parser_02`.

# 2 Програмна реалізація парсера

## 2.1 Перші кроки

Для побудови синтаксичного аналізатора методом рекурсивного спуску для кожного правила граматики визначають функцію (метод у випадку об'єктно-орієнтованого підходу), наприклад, таким чином.

1. Кожному<sup>2</sup> нетерміналу граматики з іменем `Name` поставимо у відповідність функцію `parseName()`.
2. Наявність терміналу (лексему `lexeme` з токеном `token`) перевірятимемо функцією `parseToken(lexeme, token)`.

<sup>2</sup>Крім нетерміналів, які вже були опрацьовані лексичним аналізатором (тут це `Ident`, `Float`, `Int`, `WhiteSpace`, `Newline`), та тих, які є низькорівневими допоміжними засобами для визначення інших нетерміналів, їх ще називають **фрагментами** (тут це `Letter`, `Digit`, `Blank`, `Tab`).

3. Всі оголошені функції синтаксичного аналізатора мають бути реалізовані, зокрема функції читання таблиці символів програми, обробки помилок (винятків) тощо.
4. Синтаксичний аналіз починають з запуску функції, що відповідає кореневому нетерміналу, в нашому випадку — функцію `parseProgram()` для нетермінала граматики `Program`.
5. Функції розбору читають лексеми з таблиці символів програми, і якщо це відповідні лексеми та у правильному порядку, повертають `True`. Інакше генерується помилка, для обробки якої створюють функцію, наприклад `failParse()`.

Почнемо з правила для кореневого нетермінала граматики

```
Program = program StatementList end
```

Для цього визначимо функцію `parseProgram()` через `parseToken(lexeme, token)` та `parseStatementList()`, наприклад, так:

```
Program =                                визначити parseProgram() як
    program                             parseToken('program', 'keyword')
    StatementList                       parseStatementList()
    end                                 parseToken('end', 'keyword')
```

Або, у нотації мови python, додавши код для виведення повідомлень:

```
def parseProgram():
    try:
        # перевірити наявність ключового слова 'program'
        parseToken('program', 'keyword')

        # перевірити синтаксичну коректність списку інструкцій StatementList
        parseStatementList()

        # перевірити наявність ключового слова 'end'
        parseToken('end', 'keyword')

        # повідомити про синтаксичну коректність програми
        print('Parser: Синтаксичний аналіз завершився успішно')
        return True
    except SystemExit as e:
        # Повідомити про факт виявлення помилки
        print('Parser: Аварійне завершення програми з кодом 0'.format(e))
```

Очевидно, тепер потрібно визначити функції `parseToken(lexeme, token)` та `parseStatementList()`. Почнемо з функції `parseToken(lexeme, token)`:

```
# Функція перевіряє, чи у поточному рядку таблиці розбору
# зустрілась вказана лексема lexeme з токеном token
def parseToken(lexeme,token):
    # доступ до поточного рядка таблиці розбору
    global numRow

    # збільшити відступ
    indent = nextIndt()

    # перевірити, чи є ще записи в таблиці розбору
    # len_tableOfSymb - кількість лексем (записів) у таблиці розбору
    if numRow > len_tableOfSymb :
        failParse('неочікуваний кінець програми',(lexeme,token,numRow))

    # прочитати з таблиці розбору
    # номер рядка програми, лексему та її токен
    numLine, lex, tok = getSymb()
    # тепер поточним буде наступний рядок таблиці розбору
    numRow += 1

    # чи збігаються лексема та токен таблиці розбору з заданими
    if (lex, tok) == (lexeme,token):
        # вивести у консоль номер рядка програми та лексему і токен
        print(indent+'parseToken: В рядку 0 токен 1'.format(numLine,(lexeme,token)))
        res = True
    else:
        # згенерувати помилку та інформацію про те, що
        # лексема та токен таблиці розбору (lex,tok) відрізняються від
        # очікуваних (lexeme,token)
        failParse('невідповідність токенів',(numLine,lex,tok,lexeme,token))
        res = False
    # перед поверненням - зменшити відступ
    indent = predIndt()
    return res
```

Функції `nextIndt()` та `predIndt()`, використовуються при виведенні елементів синтаксичних конструкцій у консоль, відповідно, зменшуючи або збільшуючи відступ.

```
stepIndt = 2
indt = 0
```

```
def nextIndt():
    global indt
    indt += stepIndt
    return ' '*indt
```

```
def predIndt():
    global indt
    indt -= stepIndt
    return ' '*indt
```

Змінні `numRow` та `len_tableOfSymb` визначені як глобальні змінні. Функція `getSymb()` просто повертає значення з таблиці розбору за ключем `numRow` – номером поточної лексеми (поточного запису) :

```
# Прочитати з таблиці розбору поточний запис за його номером numRow
# Повернути номер рядка програми, лексему та її токен
def getSymb():
    # таблиця розбору реалізована у формі словника (dictionary)
    # tableOfSymb = {numRow: (numLine, lexeme, token, indexOfVarOrConst)
    numLine, lexeme, token, _ = tableOfSymb[numRow]
    return numLine, lexeme, token
```

Функція `failParse(str, tuple)` обробляє помилки, виявлені в ході синтаксичного аналізу: генерує повідомлення про факт помилки, діагностичне повідомлення та код аварійного завершення парсера. Параметр `str` – змістовний "ідентифікатор" виявленої при розборі помилки.

```
# Обробити помилки
# зараз це - єдина можлива з описом 'невідповідність токенів'
def failParse(str,tuple):
    if str == 'неочікуваний кінець програми':
        (lexeme,token,numRow)=tuple
        print('Parser ERROR: \n\t Неочікуваний кінець програми -
              в таблиці символів (розбору) немає запису з номером {1}.
              \n\t Очікувалось - {0}'.format((lexeme,token),numRow))
        exit(1001)
    elif str == 'невідповідність токенів':
        (numLine,lexeme,token,lex,tok)=tuple
        print('Parser ERROR: \n\t В рядку {0} неочікуваний елемент ({1},{2}).
              \n\t Очікувався - ({3},{4}).'.format(numLine,lexeme,token,lex,tok))
        exit(1)
```

Тепер необхідно визначити функцію `parseStatementList()` у відповідності до граматики `MicroLang2`. Однак, з метою перевірки працездатності парсера для програм тривіального змісту, визначимо заглушку (`stub`) для цієї функції.

## 2.2 Заглушка для `parseStatementList()`

```
# Функція-зглушка (stub) для розбору за правилом для StatementList
# просто повідомляє про її виклик
# завжди повертає True
def parseStatementList():
```



```

# збільшити відступ
indent = nextIndt()

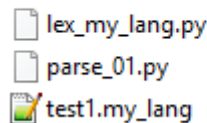
print(indent+ 'parseStatementList() - заглушка')

# перед поверненням - зменшити відступ
indent = predIndt()
return True

```

Перевіримо роботу нашого парсера, для чого перейдемо до теки `my_lang_parser\parser_01`. `StatementList=stub`.

```
<< my_lang_parser > parser_01.StatementList=stub
```



lex\_my\_lang.py  
parse\_01.py  
test1.my\_lang

Повний код парсера див. у файлі `parse_01.py`, крім визначення наведених уже функцій містить, зокрема, код запуску синтаксичного аналізатора:

```

if FSucces == ('Lexer', True):
    parseProgram()

```

Очевидно, що парсер буде запущений, тільки у випадку успішності фази лексичного аналізу.

Нехай файл `test.my_lang` містить код, коректний відносно реалізованої частини граматики:

```

1 program
2
3 end

```

Перевіримо роботу синтаксичного аналізатора у консолі:

```

> python parse_01.py
1  program      keyword
3  end          keyword
Lexer: Лексичний аналіз завершився успішно
-----
tableOfSymb:{1: (1, 'program', 'keyword', ''),
              2: (3, 'end', 'keyword', '')}
-----

parseToken: В рядку 1 токен ('program', 'keyword')
parseStatementList() - заглушка
parseToken: В рядку 3 токен ('end', 'keyword')
Parser: Синтаксичний аналіз завершився успішно

```

Лексичний аналізатор вивів таблицю розбору у консоль – вона містить дві лексеми, знайдені у рядках 1 та 3 коду в файлі `test.my_lang`. Далі виводиться та ж таблиця у формі словника `tableOfSymb`. В наступних трьох рядках – результат виклику функцій `parseToken('program', 'keyword')`, заглушки `parseStatementList()` та `parseToken('end', 'keyword')`. Останній рядок – резюме від власне `parseProgram()`. Змінимо програму у файлі `test.my_lang`

так:

```
1
2 program      end
```

Запуск парсера

```
>python parse_01.py
```

виводить результат, що свідчить про коректний синтаксис проаналізованої програми:

```
2  program      keyword
2  end          keyword
Lexer: Лексичний аналіз завершився успішно
-----
tableOfSymb:{1: (2, 'program', 'keyword', ''),
              2: (2, 'end', 'keyword', '')}
-----
  parseToken: В рядку 2 токен ('program', 'keyword')
  parseStatementList() - заглушка
  parseToken: В рядку 2 токен ('end', 'keyword')
Parser: Синтаксичний аналіз завершився успішно
```

Змінимо програму у файлі `test.my_lang`, записавши замість `program` лексему `program1`:

```
1 program1
2
3 end
```

Розбір програми свідчить про коректну роботу лексичного аналізатора – він знайшов ідентифікатор `program1` у рядку 1:

```
>python parse_01.py
1  program1      id          1
3  end          keyword
-----
tableOfSymb:{1: (1, 'program1', 'ident', 1),
              2: (3, 'end', 'keyword', '')}
-----
Lexer: Лексичний аналіз завершився успішно
```

Парсер теж коректно опрацьовує код: найперше він очікував зустріти не ідентифікатор, а ключове слово `program`, тому, при обробці виклику функції `parseToken('program', 'keyword')` був згенерований виняток та виведене діагностичне повідомлення:

```
Parser ERROR:
    В рядку 1 неочікуваний елемент (program1,id).
    Очікувався - (program,keyword).
Parser: Аварійне завершення програми з кодом 1
```

Якщо файл `test.my_lang` міститиме код:

```
1 program
```

то парсинг знову завершиться аварійно:

```
>python parse_01.py
1  program      keyword
Lexer: Лексичний аналіз завершився успішно
-----
tableOfSymb:{1: (1, 'program', 'keyword', '')}
-----
    parseToken: В рядку 1 токен ('program', 'keyword')
    parseStatementList() - заглушка
Parser ERROR:
    Неочікуваний кінець програми - в таблиці символів (розбору)
                                немає запису з номером 2.
    Очікувалось - ('end', 'keyword')
Parser: Аварійне завершення програми з кодом 1001
```

Отже виклики `parseToken('program', 'keyword')` та `parseStatementList()` завершилися успішно, а виклик `parseToken('end', 'keyword')` згенерував помилку, адже записів у таблиці розбору більше не лишилось.

Якщо файл `test.my_lang` міститиме код:

```
1
2
3 end
```

то парсинг знову завершиться аварійно:

```
>python parse_01.py
3   end           keyword
Lexer: Лексичний аналіз завершився успішно
-----
tableOfSymb:{1: (3, 'end', 'keyword', '')}
-----
Parser ERROR:
           В рядку 3 неочікуваний елемент (end,keyword).
           Очікувався - (program,keyword).
Parser: Аварійне завершення програми з кодом 1
```

Якщо програма містить код:

```
1 program
2   x := 1
3 end
```

то парсер виявить неочікуваний для себе токен `id`:

```
>python parse_01.py
1   program      keyword
2   x            id           1
2   :=          assign_op
2   1           int          1
3   end          keyword
Lexer: Лексичний аналіз завершився успішно
-----
tableOfSymb:{1: (1, 'program', 'keyword', ''),
              2: (2, 'x', 'id', 1),
              3: (2, ':=', 'assign_op', ''),
              4: (2, '1', 'int', 1),
              5: (3, 'end', 'keyword', '')}
-----
   parseToken: В рядку 1 токен ('program', 'keyword')
   parseStatementList() - заглушка
Parser ERROR:
           В рядку 2 неочікуваний елемент (x,id).
           Очікувався - (end,keyword).
Parser: Аварійне завершення програми з кодом 1
```

Парсер сприйняв ідентифікатор після лексеми `program` як синтаксичну помилку, оскільки наша функція-заклушка `parseStatementList()` нічого не аналізує і записів з таблиці розбору не читає, тому поточним записом таблиці символів програми залишається 2: (2, 'x', 'ident', 1) і виклик `parseToken('end', 'keyword')` генерує виняток (помилку).

Отже, з наведених прикладів випливає, що синтаксичний аналізатор, в межах реалізованої частини граматики, працює коректно.

Очевидно, що для повної відповідності парсера граматики мови необхідно (замість заглушки) визначити функцію `parseStatementList()`.

## 2.3 Вниз від `parseStatementList()`

Програмний код прикладу – у теці `my_lang_parser\parser_02`. Змістовно функція `parseStatementList()` має відповідати правилу

```
StatementList = Statement { Statement }
```

Правило передбачає, що список інструкцій `StatementList` – це послідовність з однієї або більшої кількості інструкцій `Statement`. Утворивши ім'я функції `parseStatement`, можемо записати у нотатції `python`:

```
def parseStatementList():
    # збільшити відступ
    indent = nextIndt()
    print(indent+'parseStatementList():')

    while parseStatement():
        pass

    # перед поверненням - зменшити відступ
    indent = predIndt()
    return True
```

Функція `parseStatementList()` викликає функцію `parseStatement()` доти, доки виклик функції `parseStatement()` повертає `True`.

Фактично з таблицею розбору працює тільки функція `parseStatement()`, і за потреби саме вона, або викликані нею функції, згенерують виняток, тому функція `parseStatementList()` не містить виклику `failParse(str,tuple)`.

Визначимо функцію `parseStatement()`. Граматики мови `MicroLang2` для нетермінала `Statement` передбачає дві альтернативи – або `Assign`, або `IfStatement`:

```
Statement = Assign | IfStatement
```

Утворимо імена функцій `parseAssign()` та `parseIf()` <sup>3</sup> для відповідних нетерміналів та подивимось, коли функція `parseStatement()` має викликати `parseAssign()`, а коли – `parseIf()`. Для цього розглянемо правила граматики для `Assign` та `IfStatement`:

```
Assign      = Ident ':= ' Expression
IfStatement = if BoolExpr then Statement else Statement endif
```

З правил видно, що інструкція присвоювання `Assign` завжди починається з лексеми, яка є ідентифікатором, а інструкція присвоювання `IfStatement` – з лексеми `if`, яка є ключовим словом. Тоді можемо записати у нотатції `python`:

<sup>3</sup>Замість імені `parseIfStatement()` тут прийнято `parseIf()`, як більш коротке і зручне.

```
def parseStatement():
    # збільшити відступ
    indent = nextIndt()
    print(indent+'parseStatement():')

    # прочитаємо поточну лексему в таблиці розбору
    numLine, lex, tok = getSymb()

    # якщо токен - ідентифікатор
    # обробити інструкцію присвоювання
    if tok == 'id':
        parseAssign()
        res = True

    # якщо лексема - ключове слово 'if'
    # обробити інструкцію розгалуження
    elif (lex, tok) == ('if', 'keyword'):
        parseIf()
        res = True

    # тут - ознака того, що всі інструкції були коректно
    # розібрані і була знайдена остання лексема програми.
    # тому parseStatement() має завершити роботу
    elif (lex, tok) == ('end', 'keyword'):
        res = False

    else:
        # жодна з інструкцій не відповідає
        # поточній лексемі у таблиці розбору,
        failParse('невідповідність інструкцій', (numLine, lex, tok, 'id або if'))
        res = False
    # перед поверненням - зменшити відступ
    indent = predIndt()
    return res
```

До функції `parseAssign()` переходимо, уже знаючи, що поточна лексема – ідентифікатор. У відповідності до правила грамматики

`Assign = Ident ':= ' Expression`

беремо цей ідентифікатор і, якщо далі зустрічається лексема `:=`, то викликаємо функцію `parseExpression()` для розбору арифметичного виразу. Відповідний код мовою python може виглядати так:

```
def parseAssign():
    # номер запису таблиці розбору
    global numRow
    # збільшити відступ
```

```

indent = nextIndt()
print(indent+'parseAssign():')

# взяти поточну лексему
numLine, lex, tok = getSymb()
print(indent+'в рядку {0} - токен {1}'.format(numLine,(lex, tok)))

# встановити номер нової поточної лексеми
numRow += 1

# print('\t'*5+'в рядку {0} - {1}'.format(numLine,(lex, tok)))
# якщо була прочитана лексема - ':'=
if parseToken(':', 'assign_op'):
    # розібрати арифметичний вираз
    parseExpression()
    res = True
else: res = False
# перед поверненням - зменшити відступ
indent = predIndt()
return res

```

Якщо визначити функцію для розбору арифметичного виразу у відповідності до правила грамматики

```

Expression = Expression ('+' | '-' | '*' | '/') Expression
            | Id
            | Const
            | '(' Expression ')'

```

яке є ліворекурсивним у першій альтернативі, то і функція `parseExpression()` буде ліворекурсивною, що при запуску програми викликатиме переповнення стека:

```

def parseExpression():
    parseExpression()
    ...

```

Очевидно, що синтаксичний аналіз методом рекурсивного спуску неможливий для ліворекурсивних правил. Тому такі правила треба трансформувати, див. напр. розд. 1.3.5 в [2], або [3, с. 81-83], або [4, с. 275-279].

## 2.4 Expression без лівої рекурсії

Замість ліворекурсивного правила для арифметичного виразу, візьмемо еквівалентне йому визначення:

```

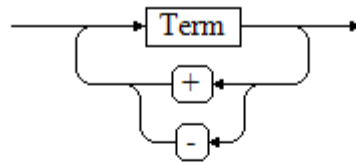
Expression = Term {('+' | '-') Term}

```

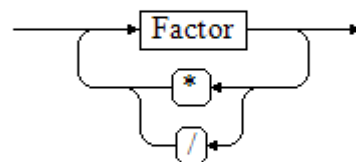
Term = Factor {('\*'|'/' ) Factor}

Factor = Id  
 | Const  
 | '(' Expression ')'

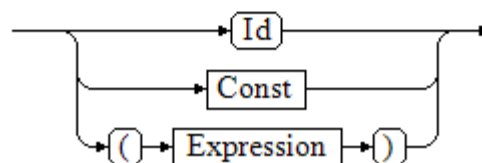
### Expression



### Term



### Factor



Правило для арифметичного виразу тепер не ліворекурсивне, і парсер для нього може бути побудований так:

```
def parseExpression():
    global numRows
    # відступ збільшити
    indent = nextIndt()
    print(indent + 'parseExpression():')
    numLine, lex, tok = getSymb()
    parseTerm()
    F = True
    # продовжувати розбирати Доданки (Term)
    # розділені лексемами '+' або '-'
    while F:
        numLine, lex, tok = getSymb()
```



```
    if tok in ('add_op'):
        numRow += 1
        print(indent+'в рядку {0} - токен {1}'.format(numLine,(lex, tok)))
        parseTerm()
    else:
        F = False
# перед поверненням - зменшити відступ
indent = predIndt()
return True

def parseTerm():
    global numRow
    # відступ збільшити
    indent = nextIndt()
    print(indent+'parseTerm():')
    parseFactor()
    F = True
# продовжувати розбирати Множники (Factor)
# розділені лексемами '*' або '/'
while F:
    numLine, lex, tok = getSymb()
    if tok in ('mult_op'):
        numRow += 1
        print(indent+'в рядку {0} - токен {1}'.format(numLine,(lex, tok)))
        parseFactor()
    else:
        F = False
# перед поверненням - зменшити відступ
indent = predIndt()
return True

def parseFactor():
    global numRow
    # відступ збільшити
    indent = nextIndt()
    print(indent+'parseFactor():')
    numLine, lex, tok = getSymb()

# перша і друга альтернативи для Factor
# якщо лексема - це константа або ідентифікатор
if tok in ('int','float','id'):
    numRow += 1
    print(indent+'в рядку {0} - токен {1}'.format(numLine,(lex, tok)))

# третя альтернатива для Factor
# якщо лексема - це відкриваюча дужка
```

```
elif lex=='(':
    numRow += 1
    parseExpression()
    parseToken(')', 'par_op')
    print(indent+'в рядку {0} - токен {1}'.format(numLine, (lex, tok)))
else:
    failParse('невідповідність у Expression.Factor',
              (numLine, lex, tok, 'rel_op, int, float, id або \'(\ ' Expression \')\')'))
# перед поверненням - зменшити відступ
indent = predIndt()
return True
```

Повний код парсера можна знайти у файлі `parse_02.py`.

## 2.5 Отже

Виконаємо синтаксичний аналіз коду у файлі `test.my_lang`:

```
1 program
2
3 v1 := (5.4 + 3)
4
5 s:=1234 + (1+x/z+3)
6 if z = x+v1*3
7   then
8     z := 1234 + ((1+x)*(z+3)) - 3+2
9   else
10    if 1234 + ((1+x)/(z+3)) < 3+2
11      then x := 1+2
12    else
13      if 1234 + (1+x/(z+3)+4) < 3+2
14        then x := 1+2
15        else z := 3/4
16      endif
17    endif
18  endif
19
20 end
```

Журнал у консолі містить понад 300 рядків, тому наводити повний протокол роботи парсера не будемо. Проте результат роботи – коректний:

```
>python parse_02.py
1  program      keyword
3  v1           id           1
3  :=          assign_op

...

Parser: Синтаксичний аналіз завершився успішно
```

У рядку 3 програми замість присвоювання поставимо знак відношення:

```
1 program
2
3 v1 < (5.4 + 3)
4
5 s:=1234 + (1+x/z+3)
6 ...
```

Тоді синтаксичний розбір приведе до результату:

```
>python parse_02.py
1  program      keyword
3  v1           id           1
3  <           rel_op
3  (           par_op
3  5.4          float        1
3  +           add_op
3  3            int          2
3  )           par_op
5  s           id           2
5  :=          assign_op

...

('len_tableOfSymb', 115)
parseToken: В рядку 1 токен ('program', 'keyword')
parseStatementList():
  parseStatement():
    parseAssign():
      в рядку 3 - токен ('v1', 'id')
Parser ERROR:
      В рядку 3 неочікуваний елемент (<,rel_op).
      Очікувався - (:=,assign_op).
Parser: Аварійне завершення програми з кодом 1
```

### 3 Оформлення та зміст звіту

Структура звіту про виконання роботи:

1. Прізвище та ім'я студента, номер групи, номер роботи – у верхньому колонтитулі. Нумерація сторінок – у нижньому колонтитулі. Титульний аркуш не потрібен.
2. На першому аркуші, угорі, – назва (тема) роботи.
3. Далі, на першому ж аркуші, – змістовна частина

Структура змістовної частини звіту

1. Коротка характеристика мови (з розділу "Вступ" специфікації розробленої мови).
2. Повна граматики мови. Остаточний її варіант — без ліворекурсивних правил.
3. Коротке пояснення про використаний лексичний аналізатор мови (зазвичай з попередньої роботи).
4. Лаконічний та змістовний опис програмної реалізації синтаксичного аналізатора.
5. План тестування, напр. як у табл. 1.

№	Тип випробування	Очікуваний результат	Кількість випробувань
1	пропуск терміналу	помилка, локалізація, діагностичне повідомлення	3
2	зайвий термінал	...	...
3	помилка у наборі ключового слова	...	...
4	помилковий оператор у виразах	...	...
5	перевірка пріоритетності операторів у виразах	...	...
6	перевірка асоціативності операторів <sup>4</sup>	...	...
7	вкладені конструкції	...	...
8	...	...	...

Табл. 1: План тестування

6. Протокол тестування. У формі текстових копій терміналу: фрагмент програми (з поясненням предмета перевірки) + фрагмент результату парсингу + Ваша оцінка результату.
7. Висновки. Тут очікується власні оцінка/констатація/враження/зауваження автора звіту — виконавця роботи.

## Література

- [1] Матеріали до виконання роботи комп'ютерного практикуму № 2 ” Лексичний аналіз методом діаграми станів”.
- [2] Стативка Ю.І. Формальні мови: Основні концепти і представлення [Текст]: навч. посіб. / Ю. І. Стативка. – Київ: КПІ ім. Ігоря Сікорського, 2023. – 87 с.
- [3] Медведєва В.М. Транслятори: лексичний та синтаксичний аналізатори [Текст]: навч.посіб. / В.М. Медведєва, В.А. Третяк. – К.: НТУУ
- [4] Aho, Alfred, Lam, Monica, Sethi, Ravi, Ullman, Jeffrey Compilers: Principles, Techniques, and Tools, 2nd edition. - Addison Wesley, 2006. - 1040 p.