**Синтаксичний аналiз**

**методом рекурсивного спуску**

Варіант № 7

Арифметика: цілі та дійсні числа, основні чотири арифметичні операції (додавання,

віднімання, ділення та множення), піднесення до степеня (правоасоціативна операція),

дужки

Особливості: експоненційна форма дійсного числа

Інструкція повторення: for (<ід>=<вираз>; <відношення>; <ід>=<вираз>){<cписок

операторів>}

Інструкція розгалуження: if <відношення> then goto <мітка>

**Повна граматика мови FuriousPeach**

Program = program ProgName DeclSection DoSection

ProgName = Ident

Ident = Letter {Letter | Digit }

DeclSection = DeclarList

DeclarList = Declaration { Declaration }

Declaration = Type ’ ’ IdenttList

IdenttList = Ident {’,’ Ident}

Type = int | real | bool

DoSection = StatementList ’bye’

StatementList = Statement {’;’ Statement }

Statement = Assign | Input | Print | ForStatement

Assign = Ident ’=’ (Expression | BoolExpr)

Expression = ArithmExpression | BoolExpr

BoolExpr = ArithmExpression Rel ArithmExpression

ArithmExpression = [Sign] Term | ArithmExpression ’+’ Term | ArithmExpression ’-’ Term

Term = Factor | Term ’\*’ Factor | Term ’/’ Factor | Term ’^’ Factor

Factor = Ident | Const | ’(’ ArithmExpression ’)’

Input = input ’(’ IdenttList ’)’

Print = print ’(’ IdenttList ’)’

ForStatement = for ’(’ IndExpr ’)’ ’{’DoBlock ’}’

IndExpr = Ident ’=’ ArithmExpression1 ’;’

BoolExpr ’;’ ArithmExpression2

ArithmExpression1 = ArithmExpression

ArithmExpression2 = Ident Add Step

DoBlock = Statement | StatementList

IfStatement = if BoolExpr ’then ’ ’goto’ Ident1 | ’else’ ’goto’ Ident2

Ident1 = Letter {Letter | Digit }

Ident2 = Letter {Letter | Digit }

Const = IntNumb | RealNumb | BoolConst

IntNumb = [Sign] UnsignedInt

RealNumb = [Sign] UnsignedReal

Sign = ’+’ | ’-’

UnsignedInt = Digit {Digit}

UnsignedReal = ’.’ UnsignedInt | UnsignedInt ’.’ | UnsignedInt ’.’ UnsignedInt

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’

BoolConst = true | false

Rel = ’==’ | ’<= ’ | ’<’ | ’>’ | ’>=’ | ’<>’

1. **Лексичний аналiзатор мови**

Лексичний аналiзатор (сканер, лексер) та таблиця символiв програми iмпортуються так:

from lex\_my\_lang import lex

from lex\_my\_lang import tableOfSymb

Виклик сканера: lex()

**Формат таблицi символiв**

Таблиця символiв програми tableOfSymb – це словник (dictionary) мови

python. Формат був визначений при розробцi лексичного аналiзатора:

{ n\_rec : (num\_line, lexeme, token, idxIdConst) }

де:

n\_rec – номер запису в таблицi символiв програми;

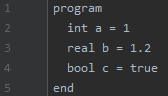
num\_line – номер рядка програми;

lexeme – лексема;

token – токен лексеми;

idxIdConst – iндекс iдентифiкатора або константи у таблицi iдентифiкаторiв та констант вiдповiдно.

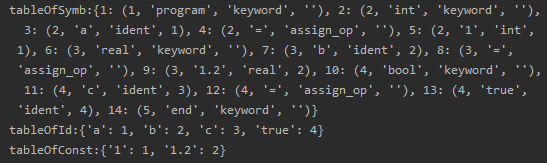
Наприклад, для програми з п’яти рядкiв:



Лексичний аналiзатор будує таблицю символiв, яка синтаксичним аналiзатором

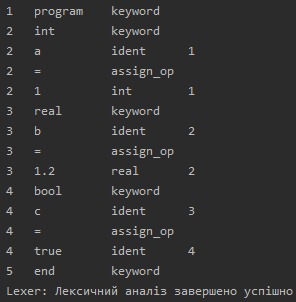
виводиться у консоль командою

print(’tableOfSymb:0’.format(tableOfSymb)):



Бiльш зручний для читання варiант виводиться лексичним аналiзатором у

форматi num\_line lexeme token idxIdConst (без номера запису у таблицi розбору n\_rec):



**Приклади програм для тестування**

Серед прикладiв програм для тестування лексичного аналiзатора є один базовий , який мiстить синтаксичнi конструкцiї, наявнi у розробленiй мовi. Всi iншi приклади мiстять код, який або демонструє бiльш складнi синтаксичнi конструкцiї, наприклад вкладенi, або мiстять синтаксичнi помилки рiзних типiв.

Тестування парсера передбачає синтаксичний розбiр прикладу коду та оцiнку правильностi його роботи (висновок про синтаксичну коректнiсть чи некоректнiсть коду, повiдомлення про знайдену помилку/помилки, її/їх локалiзацiю, дiагностичнi повiдомлення тощо).

**2. Програмна реалiзацiя парсера**

**Першi кроки**

Для побудови синтаксичного аналiзатора методом рекурсивного спуску для кожного правила граматики визначають функцiю (метод у випадку об’єктно-орiєнтованого пiдходу), наприклад, таким чином.

1. Кожному нетермiналу граматики з iменем Name поставимо у вiдповiднiсть

функцiю parseName().

2. Наявнiсть термiнала (лексеми lexeme з токеном token) перевiрятимемо

функцiєю parseToken(lexeme,token).

3. Всi оголошенi функцiї синтаксичного аналiзатора мають бути реалiзованi,

зокрема функцiї читання таблицi символiв програми, обробки помилок

(виняткiв) тощо.

4. Синтаксичний аналiз починають з запуску функцiї, що вiдповiдає кореневому нетермiналу, в нашому випадку — функцiю parseProgram() для нетермiнала граматики Program.

5. Функцiї розбору читають лексеми з таблицi символiв програми, i якщо це

вiдповiднi лексеми та у правильному порядку, повертають True. Iнакше генерується помилка, для обробки якої створюють функцiю, наприклад failParse().

Почнемо з правила для кореневого нетермiнала граматики

Program = program StatementList end

Для цього визначимо функцiю parseProgram()у нотацiї мови Python, додавши код для виведення повiдомлень:

# Функцiя для розбору за правилом

# Program = program StatementList end

# читає таблицю розбору tableOfSymb

def parseProgram():

try:

# перевiрити наявнiсть ключового слова ’program’

parseToken(’program’,’keyword’,’’)

# перевiрити синтаксичну коректнiсть

# списку iнструкцiй StatementList

parseStatementList()

# перевiрити наявнiсть ключового слова ’end’

parseToken(’end’,’keyword’,’’)

# повiдомити про синтаксичну коректнiсть програми

print(’Parser: Синтаксичний аналiз завершився успiшно’)

return True

except SystemExit as e:

# Повiдомити про факт виявлення помилки

print(’Parser: Аварiйне завершення програми з кодом {0}’.format(e))

Очевидно, тепер потрiбно визначити функцiї parseToken(lexeme, token,

indent) та parseStatementList(). Почнемо з функцiї parseToken(lexeme, token, indent), де третiй параметр indent використовується для встановлення вiдступу при виведеннi у консоль:

# Функцiя перевiряє, чи у поточному рядку таблицi розбору

# зустрiлась вказана лексема lexeme з токеном token

# параметр indent - вiдступ при виведеннi у консоль

def parseToken(lexeme,token,indent):

# доступ до поточного рядка таблицi розбору

global numRow

# якщо всi записи таблицi розбору прочитанi,

# а парсер ще не знайшов якусь лексему

if numRow > len\_tableOfSymb :

failParse(’неочiкуваний кiнець програми’,(lexeme,token,numRow))

# прочитати з таблицi розбору

# номер рядка програми, лексему та її токен

numLine, lex, tok = getSymb()

# тепер поточним буде наступний рядок таблицi розбору

numRow += 1

# чи збiгаються лексема та токен таблицi розбору (lex, tok)

# з очiкуваними (lexeme,token)

if (lex, tok) == (lexeme,token):

# вивести у консоль номер рядка програми та лексему i токен

print(indent+’parseToken:

В рядку {0} токен {1}’.format(numLine,(lexeme,token)))

return True

else:

# згенерувати помилку та iнформацiю про те, що

# лексема та токен таблицi розбору (lex,tok) вiдрiзняються вiд

# очiкуваних (lexeme,token)

failParse(’невiдповiднiсть токенiв’,(numLine,lex,tok,lexeme,token))

return False

Змiннi numRow та len\_tableOfSymb визначенi як глобальнi змiннi. Функцiя

getSymb() просто повертає значення з таблицi розбору за ключем numRow –

номером поточної лексеми (поточного запису) :

# Прочитати з таблицi розбору поточний запис за його номером numRow

# Повернути номер рядка програми, лексему та її токен

def getSymb():

# таблиця розбору реалiзована у формi словника (dictionary)

# tableOfSymb = {numRow: (numLine, lexeme, token, indexOfVarOrConst)

numLine, lexeme, token, \_ = tableOfSymb[numRow]

return numLine, lexeme, token

Функцiя failParse(str, tuple) обробляє помилки, виявленi в ходi синта-

ксичного аналiзу: генерує повiдомлення про факт помилки, дiагностичне повi-

домлення та код аварiйного завершення парсера. Параметр str – змiстовний

”iдентифiкатор” виявленої при розборi помилки.

# Обробити помилки:

# 1) ’неочiкуваний кiнець програми’

# 2) ’невiдповiднiсть токенiв’

def failParse(str,tuple):

if str == ’неочiкуваний кiнець програми’:

(lexeme,token,numRow)=tuple

print(’Parser ERROR: \n\t Неочiкуваний кiнець програми -

в таблицi символiв (розбору) немає запису з номером {1}.

\n\t Очiкувалось - {0}’.format((lexeme,token),numRow))

exit(1001)

elif str == ’невiдповiднiсть токенiв’:

(numLine,lexeme,token,lex,tok)=tuple

if lexeme == 'int' and token == 'keyword':

return ''

print(’Parser ERROR: \n\t В рядку {0} неочiкуваний елемент

({1},{2}). \n\t

Очiкувався - ({3},{4})’.format(numLine,lexeme,token,lex,tok))

exit(1)

Тепер необхiдно визначити функцiю parseStatementList() у вiдповiдностi до нашої граматики. Однак, з метою перевiрки працездатностi парсера для програм тривiального змiсту, визначимо заглушку (stub) для цiєї функцiї.

**Заглушка для parseStatementList()**

# Функцiя-заглушка (stub) для розбору за правилом для StatementList

# просто повiдомляє про її виклик

# завжди повертає True

def parseStatementList():

print(’\t parseStatementList():’)

return True

Перевiримо роботу нашого парсера.

from lex\_my\_lang import lex

from lex\_my\_lang import tableOfSymb

lex()

print(’-’\*30)

print(’tableOfSymb:{0}’.format(tableOfSymb))

print(’-’\*30)

# номер рядка таблицi розбору/лексем/символiв ПРОГРАМИ tableOfSymb

numRow=1

# кiлькiсть записiв у таблицi розбору

len\_tableOfSymb = len(tableOfSymb)

# вже розглянутi нами функцiї

parseProgram()

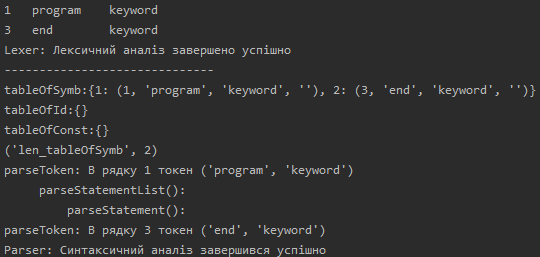
**3. Тестування синтаксичного аналізатора**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тип випробування | Очiкуванний ре зультат | Кiлькiсть ви пробувань |
| 1 | пропуск термiнала | помилка, локалiзацiя, дiагностичне повiдомлення | 4 |
| 2 | зайвий термiнал | помилка, локалiзацiя, дiагностичне повiдомлення | 1 |
| 3 | помилка у наборi ключового слова | помилка, локалiзацiя, дiагностичне повiдомлення | 1 |
| 4 | помилковий оператор у виразах | помилка, локалiзацiя, дiагностичне повiдомлення | 1 |
| 5 | вкладенi конструкцiї | аналізатор працює коректно | 1 |

Табл. 1: План тестування

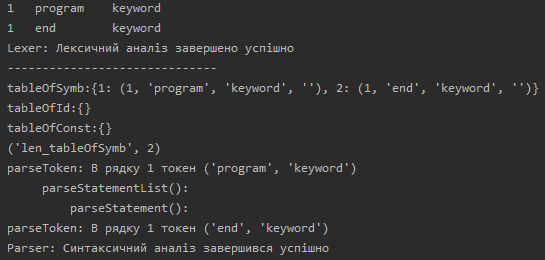


Перевiримо роботу синтаксичного аналiзатора у консолi:



Лексичний аналiзатор вивiв таблицю розбору у консоль – вона мiстить двi лексеми, знайденi у рядках 1 та 3 коду в файлi test.my\_lang. Далi виводиться та ж таблиця у формi словника tableOfSymb. В наступних рядках – результат виклику функцiй parseToken(’program’, ’keyword’, ”), parseStatementList() та parseToken(’end’, ’keyword’, ”). Останнiй рядок – резюме вiд власне parseProgram(). Змiнимо програму у файлi test.my\_lang, так:





Змiнимо програму у файлi test.my\_lang так:

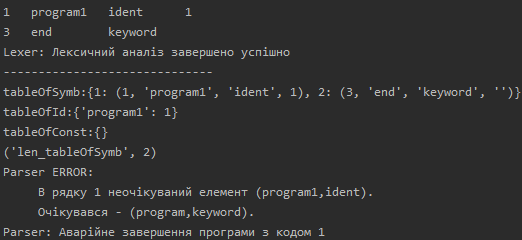


Розбiр програми свiдчить про коректну роботу лексичного аналiзатора –

вiн знайшов iдентифiкатор program1 у рядку 1.

Парсер коректно опрацьовує код: найперше вiн очiкував зустрiти не

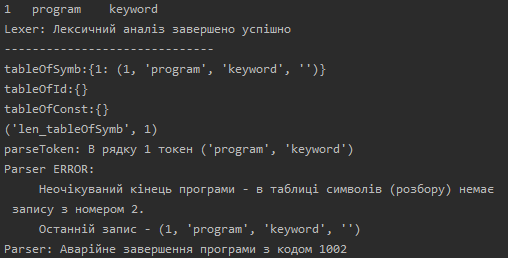
iдентифiкатор, а лексему program, тому, при обробцi виклику parseToken(’program’, ’keyword’, ”) був згенерований виняток та виведене дiагностичне повiдомлення:



Якщо файл test.my\_lang мiститиме код:



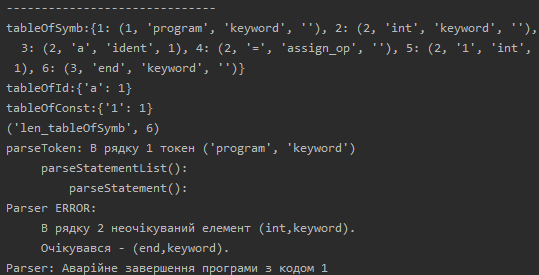
то парсинг знову завершиться аварiйно:



Розбiр коду:



виявить неочiкуваний для парсера токен ident:



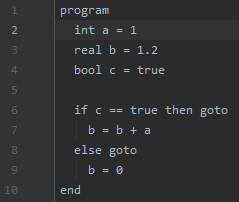
Парсер сприйняв iдентифiкатор пiсля лексеми program як синтаксичну помилку, оскiльки наша функцiя-заглушка parseStatementList() нiчого не аналiзує i записiв з таблицi розбору не читає, тому поточним записом таблицi символiв програми залишається 2: (2, ’x’, ’ident’, 1) i виклик parseToken(’end’, ’keyword’, ”) генерує виняток (помилку).

Отже, з наведених прикладiв випливає, що синтаксичний аналiзатор, в межах реалiзованої частини граматики, працює коректно.

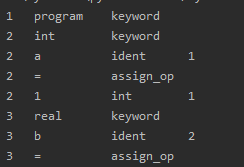
Для повної вiдповiдностi парсера граматицi мови необхiдно визначити ре-

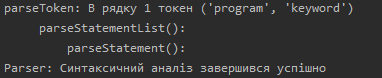
шту функцiй, починаючи з parseStatementList(), у вiдповiдностi до правил граматики мови.

Виконаємо синтаксичний аналiз коду у файлi test.my\_lang:



Таблиця символiв мiстить дуже багато записiв, тому наводити повний протокол роботи парсера не будемо. Результат роботи програми коректний:





**Висновки:** Був розроблений парсер для мови програмування FuriousPeach, розглянуто принципи його роботи з відповідними фрагментами коду. Також були проведені відповідні тестування роботи парсера, все працює як потрібно.