Круть Катерина, ТІ-01
Лабораторна робота №3

Лексичний аналіз методом діаграми станів
Варіант № 12

Арифметика: цілі та дійсні числа, основні чотири арифметичні операції (додавання, віднімання, ділення та множення), піднесення до степеня(правоасоціативна операція), дужки

Особливості: унарний мінус

Інструкція повторення: for <ід>=<вираз> by <вираз> while<відношення> do <оператор>

Інструкція розгалуження: if <відношення> {<список операторів>}

Повна граматика мови KK

Program = start ProgName ‘;’ StatementList
ProgName = Ident

Алфавіт
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’
SpecSsign = ’.’ | ’,’ | ’:’ | ’;’ | ’(’ | ’)’| ’=’ | ’+’ | ’-’ | ’*’ | ’/’ | ‘^’ | ’<’ | ’>’| WhiteSpace | EndOfLine
WhiteSpace = ’ ’ | ’\t’
EndOfLine = ’\n’ | ’\r’ | ’\r\n’ | ’\n\r’

Символи
SpecSymbols = ArithOp | RelOp | BracketsOp | AssignOp | Punct
ArithOp = AddOp | MultOp | NeltOp
AddOp = ’-’
MultOp = ’*’ | ’/’
NeltOp = ‘^’
RelOp = ’==’ | ’<= ’ | ’<’ | ’>’ | ’>=’ | ’<>’ | ‘!=’
BracketsOp = ’(’ | ’)’
AssignOp = ’=’
Punct = ’.’ | ’,’ | ’:’ | ’;’

Індифікатор
Ident = Letter {Letter | Digit }

Конастанти
Const = IntNumb | RealNumb | BoolConst

IntNumb = [Sign] UnsignedInt
RealNumb = [Sign] UnsignedReal
Sign = ’-’
UnsignedInt = Digit {Digit}
UnsignedReal = ’.’ UnsignedInt| UnsignedInt ’.’ | UnsignedInt ’.’ UnsignedInt
BoolConst = true | false

Ключові слова
KeyWords = start | read | print | for | by | while | do | if | end

Синтаксис
Expression = ArithmExpression | BoolExpr
BoolExpr = Expression RelOp Expression | true | false
ArithmExpression = [Sign] Term | ArithmExpression ’+’ Term | ArithmExpression ’-’ Term
Term = Factor | Term ’*’ Factor | Term ’/’ Factor | ‘(‘ Tern ‘^’ Factor ‘)’
Factor = Ident | Const | ’(’ ArithmExpression ’)’

Арифметичні операції
AddOp = ’+’| ’-’
MultOp = ’*’ | ’/’
NeltOp = ‘^’

Оголошення
IdenttList = Ident {’,’ Ident}

Інструкції
DoSection = StatementList
StatementList = Statement {’;’ Statement }
Statement = Assign | Inp | Out | ForStatement | IfStatment

Присвоєння
Assign = Ident ’=’ Expression

Введення
Inp = read ’(’ IdenttList ’)’.

Виведення
Out = print ’(’ IdenttList ’)’.

Інструкція повторення
ForStatement = for ‘(‘ IndExpr1 ‘)’ by ‘(‘ IndExpr2 ‘)’ while ’(‘ BoolExpr ‘)’ do ‘{‘ DoBlock ‘}’
IndExpr1 = Ident ’=’ ArithmExpression1
IndExpr2 = ArithmExpression2
DoBlock = ’{’ Statement ’}’ | ’{’ StatementList ’}’

Умовний оператор
IfStatement = if ‘(’ Relation ‘)’ ‘{‘ DoBlock ‘}’
Relation = BoolExpr

Таблиця лексем

Код	Приклади лексем	Токен	Неформальний опис
1	a, xyz1, data3	ident	ідентифікатор
2	123, 0, 521	intnum	ціле без знака
3	34.76	realnum	дійсне без знака
4	-6.7	realnum	дійсне зі знаком
5	true	boolval	логічне значення
6	start	keyword	символ start
7	read	keyword	символ read
8	print	keyword	символ print
9	for	keyword	символ for
10	by	keyword	символ by
14	do	keyword	символ do
16	while	keyword	символ while
17	if	keyword	символ if
18	end	keyword	символ end
19	false	boolval	символ false
20	=	assign_op	символ =
21	+	add_op	символ +
22	-	add_op	символ -
23	*	mult_op	символ *
24	/	mult_op	символ /
25	^	nelt_op	символ ^
26	<	rel_op	символ <
27	<=	rel_op	символ <=

28	==	rel_op	символ ===
29	>	rel_op	символ >
30	>=	rel_op	символ >=
31	!=	rel_op	символ !==
32	(	par_op	символ (
33	)	par_op	символ)
34	{	par_op	символ {
35	}	par_op	символ }
36	.	punct	символ .
37	,	punct	символ ,
38	:	punct	символ :
39	;	punct	символ ;
40	\32,\t	ws	пробільні символи
41	\n, \r, \r\n, \n\r	ls	кінець рядка

Лексичний аналізатор
Лексичний аналізатор перевіряє коректність символів програми і формує таблицю символів.

Формат таблиці символів
Таблиця розбору реалізована як словник tableOfSymb у форматі:
{ n_rec : (num_line, lexeme, token, idxIdConst) } де:
n_rec – номер запису в таблиці символів програми;
num_line – номер рядка вхідної програми;
lexeme – лексема;
token – токен лексеми;
idxIdConst – індекс ідентифікатора або константи у таблиці ідентифікаторів та констант відповідно; для інших лексем – порожній рядок.

Базовий приклад програми

```
start ProgramName
   c = 1.0 * 9.0 + 3.0;
   c = -c;
   d = 3.0 ^ (5.0 / 10.0);
   for ( I = 1 ) by ( I = I + 1) while (I < 10) do {
       print(i);
   }
   I = 1;
```

```
    I = -I;

    if ( I != 10 ) {
        print (i);
    }
    if ( I == 10 ) {
        print (i);
    }
    if ( I > 10 ) {
        print (i);
    }
    if ( I < 10 ) {
        print (i);
    }

    if ( I <= 10 ) {
        print (i);
    }
    if ( I >= 10 ) {
        print (i);
    }
end
```

Програмна реалізація

Почнемо з правила для кореневого нетермінала граматики
Program = start StatementList

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

```
def parseProgram():
    print('\n' + '=' * 60 + '\n')
    try:
        parseToken('start', 'keyword', '')
        parseToken('', 'ident', '')
        parseStatementList()
        parseToken('end', 'keyword', '')

        print('\nParser: Синтаксичний аналіз і трансляція завершені успішно')
        return True
    except SystemExit as e:
        print('\nParser: Аварійне завершення програми з кодом {0}'.format(e))
        exit()
```

перевіряємо наявність ключового слова start і назви програми, а потім списку інструкцій. При успішному аналізі виведемо відповідне повідомлення. У разі неуспішного аналізу відловлюємо помилку і завершуємо виконання.

parseToken(): Функція перевіряє, чи у поточному рядку таблиці розбору зустрілась вказана лексема lexeme з токеном token

```
def parseToken(lexeme, token, indent):
    global numRow
    if numRow > len_tableOfSymb:
        failParse('неочікуваний кінець програми', (lexeme, token, numRow))
    numLine, lex, tok = getSymb()
    numRow += 1
    if lexeme == '':
        if tok == token:
            print('*' * 30, tok, token)
            if viewSyntax:
                print(indent + 'parseToken():\n' + indent + '[{0}]:
{1}'.format(numLine, (lex, tok)))
            return True
        else:
            failParse('невідповідність токенів', (numLine, lex, tok))
            return False
    if (lex, tok) == (lexeme, token):
        if viewSyntax:
            print(indent + 'parseToken():\n' + indent + '[{0}]:
{1}'.format(numLine, (lex, tok)))
        return True
    else:
        failParse('невідповідність лексем', (numLine, lex, tok, lexeme,
token))
        return False
```

Прочитати з таблиці розбору поточний запис. Повертає номер рядка програми, лексему та її токен

```
def getSymb():
    if numRow > len_tableOfSymb:
        return False
    numLine, lexeme, token, _ = tableOfSymb[numRow]
    return numLine, lexeme, token
```

Функція parseStatementList() для розбору за правилом StatementList = Statement { Statement } викликає функцію parseStatement() доти, доки parseStatement() повертає True

```
def parseStatementList(specInstr=''):
    print('\t parseStatementList():')
    while parseStatement(specInstr):
        pass
    return True


def parseStatement(specInstr=''):
    global numRow
    if getSymb():
        numLine, lex, tok = getSymb()
    else:
        return False
    if (lex, tok) == ('end', 'keyword'):
```

```
        return False
    if viewSyntax:
        print('\t\t parseStatement():')
    if tok == 'ident':
        parseAssign()
        parseToken(';', 'punct', '\t' * 5)
        return True
    elif (lex, tok) == ('read', 'keyword'):
        parseRead()
        parseToken(';', 'punct', '\t' * 5)
        return True
    elif (lex, tok) == ('print', 'keyword'):
        parsePrint()
        parseToken(';', 'punct', '\t' * 5)
        return True
    elif (lex, tok) == ('if', 'keyword'):
        parseIf()
        return True
    elif (lex, tok) == ('for', 'keyword'):
        parseFor()
        return True
    elif specInstr == 'IF':
        if (lex, tok) == ('}', 'brackets_op'):
            return False
        else:
            failParse('невідповідність інструкцій', (numLine, lex, tok, ''))
            return False
    elif specInstr == 'FOR':
        if (lex, tok) == ('}', 'brackets_op'):
            return False
        else:
            failParse('невідповідність інструкцій', (numLine, lex, tok, ''))
            return False
    else:
        failParse('невідповідність інструкцій', (numLine, lex, tok, ''))
        return False
```

До функції parseAssign() переходимо, уже знаючи, що поточна лексема – ідентифікатор. У відповідності до правила граматики Assign = Ident ’=’ Expression
беремо цей ідентифікатор і, якщо далі зустрічається лексема =, то викликаємо функцію parseExpression() для розбору арифметичного виразу:

```
def parseAssign():
    global numRow
    if viewSyntax:
        print('\t' * 4 + 'parseAssign():')
    numLine, lex, tok = getSymb()
    postfixCode.append((lex, tok))  # Трансляція
    # if viewTranslation:
    #     configToPrint(lex, numRow)
    numRow += 1
    if viewSyntax:
        print('\t' * 5 + '[{0}]: {1}'.format(numLine, (lex, tok)))
    if parseToken('=', 'assign_op', '\t\t\t\t\t'):
        numRowCopy = numRow - 1
```

```
        parseExpression()
        postfixCode.append(('=', 'assign_op'))  # Трансляція
        # Бінарний оператор  '=' додається після своїх операндів
        # if viewTranslation:
        #     configToPrint('=', numRowCopy)
        return True
    else:
        return False
```

```
def parseIdent():
    if viewSyntax:
        print('\t' * 6 + 'parseIdent():')
    global numRow
    if numRow > len_tableOfSymb:
        failParse('неочікуваний кінець програми', ('', 'ident', numRow))
    numLine, lex, tok = getSymb()
    print('-' * 30, getSymb(), tok)
    if tok == 'ident':
        numRow += 1
        postfixCode.append((lex, tok))
        if viewSyntax:
            print('\t' * 6 + 'parseToken():\n' + '\t' * 6 + '[{0}]:
{1}'.format(numLine, (lex, tok)))
        return True
    else:
        return False
```

Правило для арифметичного виразу:

```
def parseExpression():
    global numRow, postfixCode

    if viewSyntax:
        print('\t' * 5 + 'parseExpression():')
    parseTerm()
    F = True
    while F:
        if getSymb():
            numLine, lex, tok = getSymb()
        else:
            return True
        if tok in 'add_op':
            numRowCopy = numRow
            numRow += 1
            if viewSyntax:
                print('\t' * 6 + '[{0}]: {1}'.format(numLine, (lex, tok)))
            parseTerm()
            postfixCode.append((lex, tok))  # трансляція
            # if viewTranslation:
            #     # configToPrint(lex, numRowCopy)
        else:
            F = False
    return True
```

```
def parseTerm():
    global numRow, postfixCode
    if viewSyntax:
        print('\t' * 6 + 'parseTerm():')
    parseFactor()
    F = True
    while F:
        if getSymb():
            numLine, lex, tok = getSymb()
        else:
            return True
        if tok in 'mult_op':
            numRowCopy = numRow
            numRow += 1
            if viewSyntax:
                print('\t' * 6 + '[{0}]: {1}'.format(numLine, (lex, tok)))
            parseFactor()
            postfixCode.append((lex, tok))
            # if viewTranslation:
            #     # configToPrint(lex, numRowCopy)
        else:
            F = False
    return True


def parseFactor():
    global numRow, postfixCode
    if viewSyntax:
        print('\t' * 7 + 'parseFactor():')
    if getSymb():
        numLine, lex, tok = getSymb()
    else:
        return True
    if viewSyntax:
        print('\t' * 7 + '[{0}]: {1}'.format(numLine, (lex, tok)))
    if tok in ('integer', 'real', 'ident'):
        postfixCode.append((lex, tok))
        # if viewTranslation:
        #     # configToPrint(lex, numRow)
        numRow += 1
        if getSymb():
            numLine, lex, tok = getSymb()
        else:
            return True
        if lex == '^':
            numRowCopy = numRow
            numRow += 1
            if viewSyntax:
                print('\t' * 6 + '[{0}]: {1}'.format(numLine, (lex, tok)))
            parseFactor()
            postfixCode.append((lex, tok))
            # if viewTranslation:
            #     # configToPrint(lex, numRowCopy)
    elif lex == '(':
        numRow += 1
        parseExpression()
        parseToken(')', 'brackets_op', '\t' * 7)
```

```
    elif lex == '-':
        numRowCopy = numRow
        numRow += 1
        parseExpression()
        postfixCode.append(('NEG', tok))
        # if viewTranslation:
        #     # configToPrint(lex, numRowCopy)
    else:
        failParse('невідповідність у Expression.Factor',
                  (numLine, lex, tok, 'integer, real, ident або \'(\'
Expression \')\''))
    return True
```

обробка оператора читання:

```
def parseRead():
    global numRow
    if viewSyntax:
        print('\t' * 4 + 'parseRead():')
    _, lex, tok = getSymb()
    if lex == 'read' and tok == 'keyword':
        numRow += 1
        parseToken('(', 'brackets_op', '\t' * 5)
        parseIdentList()
        parseToken(')', 'brackets_op', '\t' * 5)
        return True
    else:
        return False
```

обробка оператора виводу:

```
def parsePrint():
    global numRow
    if viewSyntax:
        print('\t' * 4 + 'parsePrint():')
    _, lex, tok = getSymb()
    if lex == 'print' and tok == 'keyword':
        numRow += 1
        parseToken('(', 'brackets_op', '\t' * 5)
        parseExpressionList()
        parseToken(')', 'brackets_op', '\t' * 5)
        return True
    else:
        return False
```

Обробка списку виразів:

```
def parseExpressionList():
    if viewSyntax:
        print('\t' * 5 + 'parseExpressionList():')
    while parseExpression():
        if getSymb():
            numLine, lex, tok = getSymb()
        else:
            return True
        if lex == ')':
            break
```

```
        parseToken(',', 'punct', '\t\t\t\t\t')
        postfixCode.append(('OUT', 'out'))
    return True
```

розбір інструкції розгалудження за правилом IfStatement = if ‘(‘ Relation ‘)’ ‘{‘ StatementList ‘}’

```
def parseIf():
    global numRow
    if viewSyntax:
        print('\t' * 4 + 'parseIf():')
    _, lex, tok = getSymb()
    if lex == 'if' and tok == 'keyword':
        numRow += 1
        parseToken('(', 'brackets_op', '\t' * 5)
        parseBoolExpr()
        parseToken(')', 'brackets_op', '\t' * 5)
        parseToken('{', 'brackets_op', '\t' * 5)
        parseStatementList('IF')
        parseToken('}', 'brackets_op', '\t' * 5)
        return True
    else:
        return False
```

розбір інструкції повторювання за правилом ForStatement = for ‘(‘ IndExpr1 ‘)’ by ‘(‘ IndExpr2 ‘)’ while ’(‘ BoolExpr ‘)’ do ‘{‘ DoBlock ‘}’

```
def parseFor():
    global numRow
    if viewSyntax:
        print('\t' * 4 + 'parseFor():')
    _, lex, tok = getSymb()
    if lex == 'for' and tok == 'keyword':
        numRow += 1
        numLine, lex, tok = getSymb()

        parseToken('(', 'brackets_op', '\t' * 5)
        parseIdent()
        parseToken('=', 'assign_op', '\t' * 5)
        parseExpression()
        parseToken(')', 'brackets_op', '\t' * 5)
        parseToken('by', 'keyword', '\t' * 5)
        parseToken('(', 'brackets_op', '\t' * 5)
        parseIdent()
        parseToken('=', 'assign_op', '\t' * 5)
        parseExpression()
        parseToken(')', 'brackets_op', '\t' * 5)
        parseToken('while', 'keyword', '\t' * 5)
        parseToken('(', 'brackets_op', '\t' * 5)
        parseBoolExpr()
        parseToken(')', 'brackets_op', '\t' * 5)
        parseToken('do', 'keyword', '\t' * 5)
        parseToken('{', 'brackets_op', '\t' * 5)
        parseStatementList('FOR')
        parseToken('}', 'brackets_op', '\t' * 5)
        return True
    else:
        return False
```

розбір логічного виразу за правилом BoolExpr = Expression ('='|'<='|'>='|'<'|'>'|'<>') Expression

```
def parseBoolExpr():
    global numRow
    if viewSyntax:
        print('\t' * 5 + 'parseBoolExpression():')
    parseExpression()
    if getSymb():
        numLine, lex, tok = getSymb()
    else:
        return True
    numRow += 1
    parseExpression()
    if tok in ('rel_op'):
        postfixCode.append((lex, tok))  # Трансляція
        if viewSyntax:
            print('\t' * 5 + '[{0}]: {1}'.format(numLine, (lex, tok)))
    else:
        failParse('невідповідність у BoolExpr', (numLine, lex, tok, '== != <=
>= < >'))
    return True
```

Тестування

1. Пропуск термінала

a. Пропуск назви програми (Error 111)

Код:

```
start end
   d = 3.0 ^ (5.0 / 10.0);
   if ( d != 10 ) {
       print (i);
   }
end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')

Parser ERROR:
  [1]: Неочікуваний елемент ('for', keyword).
  Очікувався ідентифікатор

Parser: Аварійне завершення програми з кодом 111
```

b. Пропуск назви змінної (Error 108)

Код:

```
start ProgramName
   = 1.0
end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')
parseToken():
[1]: ('ProgramName', 'ident')
   parseStatementList():
    parseStatement():

Parser ERROR:
  [2]: Неочікуваний елемент ('=', assign_op).
  Очікувалася інструкція.

Parser: Аварійне завершення програми з кодом 108
```

c. Пропуск оператора присвоювання (Error 107)

Код:

```
start ProgramName
   c  1.5
end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')
parseToken():
[1]: ('ProgramName', 'ident')
   parseStatementList():
    parseStatement():
                  parseAssign():
                          [2]: ('c', 'ident')

Parser ERROR:
  [2]: Неочікуваний елемент ('1.5', real).
  Очікувався - ('=', assign_op).

Parser: Аварійне завершення програми з кодом 107
```

d. Пропуск закриваючої дужки (Error 107)

Код:

```
start ProgramName
   k = 1;
   if (k != 10) {
       print (k);

end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')
parseToken():
[1]: ('ProgramName', 'ident')
   parseStatementList():
      parseStatement():
           parseAssign():
              [2]: ('k', 'ident')
              parseToken():
              [2]: ('=', 'assign_op')
              parseExpression():
                 parseTerm():
                    parseFactor():
                    [2]: ('1', 'integer')
              parseToken():
              [2]: (';', 'punct')
      parseStatement():
           parseIf():
              parseToken():
              [3]: ('(', 'brackets_op')
```

```
                parseBoolExpression():
                parseExpression():
                   parseTerm():
                      parseFactor():
                      [3]: ('k', 'ident')
                parseExpression():
                   parseTerm():
                      parseFactor():
                      [3]: ('10', 'integer')
                [3]: ('!=', 'rel_op')
                parseToken():
                [3]: (')', 'brackets_op')
                parseToken():
                [3]: ('{', 'brackets_op')
   parseStatementList():
      parseStatement():
           parsePrint():
              parseToken():
              [4]: ('(', 'brackets_op')
              parseExpressionList():
              parseExpression():
                 parseTerm():
                    parseFactor():
                    [4]: ('k', 'ident')
              parseToken():
              [4]: (')', 'brackets_op')
              parseToken():
              [4]: (';', 'punct')

Parser ERROR:
  [6]: Неочікуваний елемент ('end', keyword).
  Очікувався - ('}', brackets_op).

Parser: Аварійне завершення програми з кодом 107
```

e. Пропуск знака пунктуації ; (Error 107)

Код:

```
start ProgramName
   k = 1
   print (k);
end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')
parseToken():
[1]: ('ProgramName', 'ident')
   parseStatementList():
      parseStatement():
           parseAssign():
              [2]: ('k', 'ident')
              parseToken():
              [2]: ('=', 'assign_op')
              parseExpression():
```

```
                parseTerm():
                   parseFactor():
                   [2]: ('1', 'integer')

Parser ERROR:
  [3]: Неочікуваний елемент ('print', keyword).
  Очікувався - (';', punct).

Parser: Аварійне завершення програми з кодом 107
```

2. Зайвий термінал

a. Зайвий ідентифікатор (Error 107)

Код:

```
start ProgramName bla
   print(5);
end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')
parseToken():
[1]: ('ProgramName', 'ident')
   parseStatementList():
      parseStatement():
           parseAssign():
              [1]: ('bla', 'ident')

Parser ERROR:
  [2]: Неочікуваний елемент ('print', keyword).
  Очікувався - ('=', assign_op).

Parser: Аварійне завершення програми з кодом 107
```

b. Зайвий знак (Error 109)

Код:

```
start ProgramName
   k = 1 + / 3;
end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')
parseToken():
[1]: ('ProgramName', 'ident')
   parseStatementList():
      parseStatement():
           parseAssign():
              [2]: ('k', 'ident')
              parseToken():
              [2]: ('=', 'assign_op')
              parseExpression():
                 parseTerm():
```

```
                    parseFactor():
                       [2]: ('1', 'integer')
                 [2]: ('+', 'add_op')
                 parseTerm():
                    parseFactor():
                    [2]: ('/', 'mult_op')

Parser ERROR:
  [2]: Неочікуваний елемент ('/', mult_op).
  Очікувався - 'integer, real, ident або '(' Expression ')''.

Parser: Аварійне завершення програми з кодом 109
```

c. Помилка у наборі ключового слова

Код:

```
start ProgramName
   red(k);
end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')
parseToken():
[1]: ('ProgramName', 'ident')
   parseStatementList():
      parseStatement():
           parseAssign():
              [2]: ('red', 'ident')

Parser ERROR:
  [2]: Неочікуваний елемент ('(', brackets_op).
  Очікувався - ('=', assign_op).

Parser: Аварійне завершення програми з кодом 107
```

3. Порожній файл

Parser ERROR:

```
Parser ERROR:
  Неочікуваний кінець програми - в таблиці символів (розбору) немає
запису з номером 1.
  Очікувалось - ('start', 'keyword')

Parser: Аварійне завершення програми з кодом 106
```

4. Shorted basic example without errors

Код:

```
start ProgramName
   c = 1.0 * 9.0;
   for (i = 1) by (i = i + 1) while (i < 10) do {
       print(i);
   }
   if (c != 10) {
       print (i);
   }
end
```

Результат:

```
parseToken():
[1]: ('start', 'keyword')
parseToken():
[1]: ('ProgramName', 'ident')
   parseStatementList():
      parseStatement():
           parseAssign():
              [2]: ('c', 'ident')
              parseToken():
              [2]: ('=', 'assign_op')
              parseExpression():
                 parseTerm():
                    parseFactor():
                    [2]: ('1.0', 'real')
                 [2]: ('*', 'mult_op')
                    parseFactor():
                    [2]: ('9.0', 'real')
              parseToken():
              [2]: (';', 'punct')
      parseStatement():
           parseFor():
              parseToken():
              [3]: ('(', 'brackets_op')
                 parseIdent():
                 parseToken():
                 [3]: ('i', 'ident')
              parseToken():
              [3]: ('=', 'assign_op')
              parseExpression():
                 parseTerm():
                    parseFactor():
                    [3]: ('1', 'integer')
              parseToken():
              [3]: (')', 'brackets_op')
              parseToken():
              [3]: ('by', 'keyword')
              parseToken():
              [3]: ('(', 'brackets_op')
                 parseIdent():
                 parseToken():
                 [3]: ('i', 'ident')
              parseToken():
              [3]: ('=', 'assign_op')
```

```
            parseExpression():
               parseTerm():
                  parseFactor():
                  [3]: ('i', 'ident')
               [3]: ('+', 'add_op')
               parseTerm():
                  parseFactor():
                  [3]: ('1', 'integer')
            parseToken():
            [3]: (')', 'brackets_op')
            parseToken():
            [3]: ('while', 'keyword')
            parseToken():
            [3]: ('(', 'brackets_op')
            parseBoolExpression():
            parseExpression():
               parseTerm():
                  parseFactor():
                  [3]: ('i', 'ident')
            parseExpression():
               parseTerm():
                  parseFactor():
                  [3]: ('10', 'integer')
            [3]: ('<', 'rel_op')
            parseToken():
            [3]: (')', 'brackets_op')
            parseToken():
            [3]: ('do', 'keyword')
            parseToken():
            [3]: ('{', 'brackets_op')
parseStatementList():
   parseStatement():
        parsePrint():
            parseToken():
            [4]: ('(', 'brackets_op')
            parseExpressionList():
            parseExpression():
               parseTerm():
                  parseFactor():
                  [4]: ('i', 'ident')
            parseToken():
            [4]: (')', 'brackets_op')
            parseToken():
            [4]: (';', 'punct')
   parseStatement():
            parseToken():
            [5]: ('}', 'brackets_op')
   parseStatement():
        parseIf():
            parseToken():
            [6]: ('(', 'brackets_op')
            parseBoolExpression():
            parseExpression():
               parseTerm():
                  parseFactor():
                  [6]: ('c', 'ident')
            parseExpression():
               parseTerm():
```

```
                    parseFactor():
                    [6]: ('10', 'integer')
              [6]: ('!=', 'rel_op')
              parseToken():
              [6]: (')', 'brackets_op')
              parseToken():
              [6]: ('{', 'brackets_op')
   parseStatementList():
      parseStatement():
           parsePrint():
              parseToken():
              [7]: ('(', 'brackets_op')
              parseExpressionList():
              parseExpression():
                 parseTerm():
                    parseFactor():
                    [7]: ('i', 'ident')
              parseToken():
              [7]: (')', 'brackets_op')
              parseToken():
              [7]: (';', 'punct')
      parseStatement():
              parseToken():
              [8]: ('}', 'brackets_op')
parseToken():
[9]: ('end', 'keyword')

Parser: Синтаксичний аналіз і трансляція завершені успішно
```

Висновок: в ході даної роботи я освоїла основи розробки синтаксичного аналізатору та задіяла їх на практиці. Також мною було розроблено синтаксичний аналізатор для мови програмування KK.