Матеріали до виконання лабораторної роботи № 3 Трансляція у ПОЛІЗ арифметичних виразів та інтерпретація постфіксного коду

Юрій Стативка

Квітень, 2020 р.

Зміст

Вступ

Цей текст підготовлений з огляду на спричинену карантином відсутність лекційних занять як таких. Більша частина цього тексту – послідовна розповідь про побудову транслятора арифметичних виразів у постфіксну форму (ПОЛІЗ) та інтерпретатора отриманого коду. Під транслятором розумітимемо програму, яка перекладає з вхідної мови на цільову, тут – у ПОЛІЗ. Код двох розглянутих прикладів, – транслятора та інтерпретатора арифметичних виразів, додаються у архіві.

Ті, хто вже зрозуміли як побудувати такі транслятор та інтерпретатор чи вже їх побудували, можуть тільки переглянути вимоги до оформлення звіту у розділі 4 .

Мета лабораторної роботи – програмна реалізація для розробленої мови програмування транслятора та інтерпретатора арифметичних виразів та інструкцій присвоювання. Цільова мова трансляції – ПОЛІЗ.

Постфіксна нотація (ПОЛІЗ)

Польський інверсний запис (ПОЛІЗ, польська інверсна нотація) – постфіксна нотація для операторного представлення програм. Детальний опис для арифметичних, логічних виразів та для різноманітних синтаксичних конструкцій мови програмування див. напр. [1, розд. 1.1]. Розглянемо приклади у табл. 1:

Неважко бачити, що вираз у довільній (інфіксній, префіксній, постфіксній) нотації може бути записаний у постфіксній. Інструкція присвоювання, рядок № 7, теж вважається інфіксним виразом з оператором `:=` і перекладається за загальними правилами. У рядку № 5 унарний мінус теж записується після свого аргумента, але щоб відрізняти його від оператора віднімання тут він позначений символом `NEG`, а, наприклад в [1, розд. 1.1] – символом `@`. У випадку обчислення

№	Оператор	Вираз	Постфіксна форма
1	Інфіксний	a + b	a b +
2	Інфіксний	5 - 7	5 7 -
3	Інфіксний	a * 3	a 3 *
4	Інфіксний	2 / 4	2 4 /
5	Префіксний	-9	9 NEG
6	Постфіксний	5!	5 FACT
7	Інфіксний*	x := 1.1	x 1.1 :=

Табл. 1: Приклади запису арифметичних виразів

факторіала числа з постфіксним оператором `!`, переклад зводиться до переписування виразу, можливо, – на власний розсуд, з вибором іншого позначення оператора, як-от тут це – `FACT`.

Загальна ідея – код програми вважається послідовністю операторів та операндів. Виходячи з семантики початкової мови кожному оператору призначають арність, пріоритет та асоціативність.

Наприклад адитивні арифметичні оператори `+` та `-` — визначають як двомісні (бінарні, арності 2), лівоасоціативні оператори з пріоритетом, меншим за пріоритет мультиплікативних арифметичних операторів `*` та `/`, які теж двомісні та лівоасоціативні. Останні мають менший пріоритет ніж у арифметичного оператора піднесення до степеня, позначимо його як `^`, який є двомісним та правоасоціативним. Такі характеристики відповідають семантиці арифметичних виразів, користуючись якими вираз

```
1 + 2^3^2 - 4/5 - 6*2
```

обчислюють у такому порядку:

```
(((1+(2^(3^2)))-(4/5))-(6*2)),
```

Порівнявши інфіксну (звичайну) та постфіксну форму цього виразу

```
1 + 2^3^2 - 4/5 - 6*2
```

```
1 2 3 2 ^ ^ + 4 5 / - 6 2 * -
```

бачимо, що порядок розташування операндів однаковий, а порядок операторів, зрозуміло, змінився, – адже постфіксна нотація не містить дужок. Алгоритм обчислення значення виразу у постфіксній нотації загальновідомий і наведений, зокрема, у [1, розд. 1.1.1].

Підхід, коли кожен символ програми, який не є елементом даних, вважається оператором з призначенням йому відповідних атрибутів, дуже докладно розглянуто у [1, розд. 1.1]. Проте існують і інші підходи, один з яких – рекурсивний, розглядається далі.

Метод трансляції у ПОЛІЗ

Позначимо постфіксну форму виразу `E` через `ПОЛІЗ(E)`. Тоді, наприклад, якщо `E = 'a + b'`, то `ПОЛІЗ(E) = 'a b +'`, а якщо `E = 'a := b'`, то `ПОЛІЗ(E) = 'a b :='`.

З такими позначеннями постфіксна нотація може бути визначена так:
Рекурсивне визначення ПОЛІЗу

1. Якщо `E` – ідентифікатор або константа, то `ПОЛІЗ(E) = E`.
2. Якщо `E = 'E1 op E2'`, де `op` – інфіксний оператор, то `ПОЛІЗ(E) ='ПОЛІЗ(E1) ПОЛІЗ(E2) op'`.
3. Якщо `E = 'op E1'`, де `op` – префіксний оператор, то `ПОЛІЗ(E) ='ПОЛІЗ(E1) op'`.
4. Якщо `E = 'E1 op'`, де `op` – постфіксний оператор, то `ПОЛІЗ(E) ='ПОЛІЗ(E1) op'`.
5. Якщо `E = '( E1 )'`, то `ПОЛІЗ(E) ='ПОЛІЗ(E1)'`, тобто дужки ігноруються.

Наприклад, хай `E = '1 + 2 * (3 - 4)'`, тоді за правилом 2:
`ПОЛІЗ(E) = ПОЛІЗ('1') ПОЛІЗ('2 * (3 - 4)') '+'`.
Оскільки за правилом 2:
`ПОЛІЗ('2 * (3 - 4)') = ПОЛІЗ('2') ПОЛІЗ('(3 - 4)') '*'`
та, за правилами 5 і 2:
`ПОЛІЗ('(3 - 4)') = ПОЛІЗ('3 - 4') = ПОЛІЗ('3') ПОЛІЗ('4') '-'`,
то
`ПОЛІЗ('2 * (3 - 4)') = ПОЛІЗ('2') ПОЛІЗ('3') ПОЛІЗ('4') '-' '*'`,
та
`ПОЛІЗ(E) = ПОЛІЗ('1') ПОЛІЗ('2') ПОЛІЗ('3') ПОЛІЗ('4') '-' '*' '+'`.
Насамкінець, за правилом 1:
`ПОЛІЗ(E) = '1 2 3 4 - * +'`.

При розгляді і наведеного формального визначення, і прикладу впадає у вічі аналогія з синтаксичним аналізом методом рекурсивного спуску. Відповідність ця справді є, і тому далі розглядається побудова транслятора арифметичних виразів шляхом додавання певних (семантичних) процедур до синтаксичного аналізатора (лабораторна робота № 2 першого семестру).

Метод інтерпретації

У цій лабораторній роботі програма – це послідовність операторів присвоювання змінним значень арифметичних виразів. Вихід транслятора – програма, записана у постфіксній формі – ПОЛІЗ-програма. Інтерпретатор виконує ПОЛІЗ-програму.

В загальних рисах алгоритм обчислення значення виразу, представленого у постфіксній формі, виглядає так:

Вхід: послідовність ідентифікаторів змінних, констант та операторів – арифметичних і присвоювання значення (ПОЛІЗ-програма).
Пам'ять: таблиця ідентифікаторів/змінних.
Вихід: новий стан пам'яті (нові значеня у таблиці ідентифікаторів/змінних).

1. Якщо на вході – ідентифікатор або константа, то – покласти на стек .

2. Якщо на вході двомісний оператор, то
   - зняти з вершини стека правий операнд;
   - зняти з вершини стека лівий операнд;
   - якщо оператор – арифметичний, то виконати операцію та покласти результат на стек;
   - якщо оператор – присвєння, то занести значення правого опернда у таблицю ідентифікаторів як значення змінної з ідентифікатором – лівим операндом.

3. Якщо на вході одномісний (унарний) оператор, то
   - зняти з вершини стека операнд;
   - виконати операцію та покласти результат на стек.

4. Якщо оброблений останній елемент ПОЛІЗ-програми і стек порожній, то інтерпретація завершилась успіхом, інакше – аварійне завершення.

Необхідні програми та дані

Для виконання роботи потрібні:

1. Граматика мови.

2. Специфікація арифметичних виразів розробленої мови (типи, константи, ідентифікатори, оператори тощо).

3. Лексичний аналізатор.

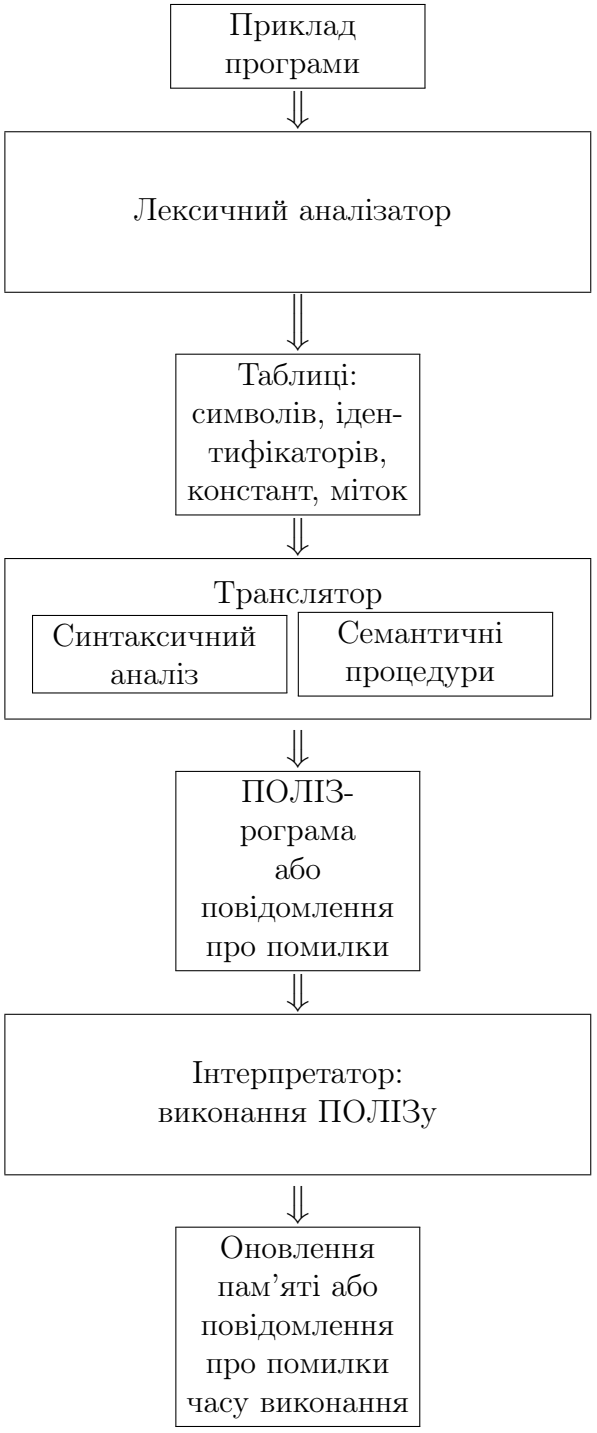
4. Синтаксичний аналізатор.

5. Формат таблиць: символів, ідентифікаторів, констант, міток.

6. Приклади програм для тестування транслятора та інтерпретатора.

Обидва розглянуті далі приклади можна знайти у теці `postfixExpr`. Перший – у `postfixExpr\postfixExpr_translator`, другий – у `postfixExpr\postfixExpr_interpreter`, з прикладами програмної реалізації транслятора та інтерпретатора відповідно.

Загальна схема взаємодії модулів інтерпретатора показані на представленій тут схемі:



1 Початкові дані для роботи

1.1 Граматика мови

Правила граматики, суттєві для розробки транслятора:

```
Program = program StatementList end

StatementList = Statement  { Statement }
```

```
Statement = Assign

Assign = Ident ':=' Expression

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

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

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

Const = Float | Int
```

Правила граматики для нетерміналів, суттєвих тільки для лексичного аналізатора:

```
Ident = Letter {Letter | Digit }

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'
```

Можна було б спростити правила

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

до

```
StatementList = Assign  { Assign },
```

але ми не станемо цього робити з міркувань мінімізації змін наявного синтаксичного аналізатора та можливості подальшого розвитку транслятора.

1.2 Специфікація арифметики

1.2.1 Типи

Мова *my_lang* підтримує значення двох арифметичних типів: `int` та `float`.

1. Цілий тип `int` може бути представлений константою `Int`, або змінною.
2. Дійсний тип `float` може бути представлений константою `Float`, або змінною.

1.2.2 Константи

Семантика

1. Кожна константа має тип, визначений її формою.

Приклади

1. `12, 234, 1.54, 34.567, 23.5`

1.2.3 Вирази

Опис

1. Вираз - це послідовність операторів і операндів, що визначає порядок обчислення значення.
2. Розрізняються арифметичні та логічні вирази.
3. Значення, обчислене за арифметичним виразом, має тип `float` або `int`.
4. Значення, обчислене за логічним виразом, має тип `boolean`.
5. Всі бінарні оператори у виразах цієї мови лівоасоціативні.

Обмеження

1. Використання неоголошеної змінної, викликає помилку на етапі трансляції або часу виконання.
2. Використання змінної, що не набула значення, викликає помилку[1].

Семантика

1. Кожна константа має тип, визначений її формою та значенням.
2. Змінна набуває значення в інструкції присвоювання `Assign` або в інструкції введення `Inp`.

Приклади

1. Factor:

[1] Можливі варіанти — фаза трансляції та/або часу виконання

```
x, 12, (a + 234)
```

2. Term:

```
m*z, 32/(b + 786)
```

3. Expression:

```
b, f1 + g, (c - 24), 0.145*8.7 + 32/(b + 786)
```

1.2.4 Арифметичні оператори

В мові *my_lang* всі арифметичні оператори інфіксні: адитивні – `'+'` та `'-'`, мультиплікаттивні – `'*'` та `'/'`.
Семантика

1. Типи обох операндів мають бути однаковим, інакше генерується виключення (помилка).
2. Тип результату збігається з типом операндів.
3. Ділення на нуль викликає помилку.
4. Ділення цілих дає цілий результат, отриманий відкиданням дробової частини від ділення тих же чисел дійсного типу.

Приклад

1. 1.234*x1/45.67, 53-34+6, 7/8

1.2.5 Іінструкція присвоювання

Опис

1. Значення, які можуть використовуватись у лівій та правій частинах інструкції присвоювання називають `l-значенням` та `r-значенням` (або `lvalue` та `rvalue`, або `left-value` та `right-value` тощо).

Семантика

1. `l-значення` має тип вказівника на місце зберігання значення змінної з ідентифікатором `Ident`.
2. `r-значення` має тип значення, обчисленого за виразом `Expression`.
3. При присвоюванні за адресою `lvalue` зберігається значення типу `rvalue`.

Приклад

1. f5 := 3/4+1.23, b := 2 * a + 3 - 7 div 5

1.3 Лексичний аналізатор вхідної мови

Використовується приклад лексичного аналізатора до лабораторної роботи №1, реалізований мовою python, див. файл `lex_my_lang_03.py`. Лексичний аналізатор (сканер, лексер) та таблиця символів програми імпортуються так:

```
from  lex_my_lang_03 import lex, tableToPrint
from  lex_my_lang_03 import tableOfSymb, tableOfId, tableOfConst, \
                            tableOfLabel, sourceCode, FSuccess
```

Виклик сканера:
`lex()`

1.4 Синтаксичний аналізатор вхідної мови

Використовується приклад синтаксичного аналізатора до лабораторної роботи №2, реалізований мовою python. Після його доповнення семантичними процедурами, він буде здійснювати одночасно синтаксичний аналіз та трансляцію, див. файл `postfixExpr_translator.py` у однойменній теці.

1.5 Формат таблиць

Формат усіх таблиць визначений при розробці лексичного аналізатора. Таблиці реалізовані як словники (`dictionary`) мови `python`.

Таблиця міток тут не використовується, проте формат наводиться.

1.5.1 Таблиця символів

Таблиця символів `tableOfSymb` :

```
{ 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.

1.5.2 Таблиця ідентифікаторів

Таблиця ідентифікаторів `tableOfId`:

```
 { Id: (idxId, type, val) }
```

де:
`Id` – ідентифікатор (лексема);
`idxId` – індекс ідентифікатора у таблиці ідентифікаторів;
`type` – тип значення змінної з ідентифікатором `Id`, при (першому) занесенні до таблиці тип вважається невизначеним `type_undef`;
`val` – значення змінної з ідентифікатором `Id`, при (першому) занесенні до таблиці значення вважається невизначеним `val_undef`.

1.5.3 Таблиця констант

Таблиця констант `tableOfConst`:

```
{ Const: (idxConst, type, val) }
```

де:
`Const` –константа (лексема);
`idxConst` – індекс константи у таблиці констант;
`type` – тип константи;
`val` – значення константи.

1.5.4 Таблиця міток

Таблиця міток `tableOfLabel`:

```
{ Label : val) }
```

де:
`Label` –мітка (лексема);
`val` – значення мітки.

1.5.5 Приклад

Для програми з п'яти рядків:

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

будуються таблиці, які можна переглянути у консолі за допомогою команд:

```
print('tableOfSymb:{0}'.format(tableOfSymb))
print('tableOfSymb:{0}'.format(tableOfId))
print('tableOfSymb:{0}'.format(tableOfConst))
print('tableOfSymb:{0}'.format(tableOfLabel))
```

Результат має приблизно такий вигляд:

```
tableOfSymb:{1: (1, 'program', 'keyword', ''),
2: (3, 'v1', 'ident', 1),
3: (3, ':=', 'assign_op', ''),
4: (3, '(', 'par_op', ''), 5: (3, '5.4', 'float', 1),
6: (3, '+', 'add_op', ''), 7: (3, '3', 'int', 2),
8: (3, ')', 'par_op', ''), 9: (3, '/', 'mult_op', ''),
10: (3, 'a', 'ident', 2), 11: (5, 'end', 'keyword', '')}

tableOfId:{'v1': (1, 'type_undef', 'val_undef'),
'a': (2, 'type_undef', 'val_undef')}

tableOfConst:{'5.4': (1, 'float', 5.4), '3': (2, 'int', 3)}

tableOfLabel:{}
```

Більш зручний для читання варіант виводиться функцією `tableToPrint('All')`:

```
 Таблиця символів
numRec     numLine    lexeme     token      index
1          1          program    keyword
2          3          v1         ident      1
3          3          :=         assign_op
4          3          (          par_op
5          3          5.4        float      1
6          3          +          add_op
7          3          3          int        2
8          3          )          par_op
9          3          /          mult_op
10         3          a          ident      2
11         5          end        keyword

 Таблиця ідентифікаторів
Ident      Type            Value           Index
v1         type_undef      val_undef       1
a          type_undef      val_undef       2

 Таблиця констант
Const      Type       Value      Index
5.4        float      5.4        1
3          int        3          2

 Таблиця міток - порожня
```

Зауважимо, що індекси ідентифікаторів та констант виводяться у консолі в останній позиції. Таблиці та функція `tableToPrint('All')` визначені у сканері

та імпортуються до інших модулів.

1.6 Приклади програм для тестування

Серед прикладів програм для тестування довільного модуля має бути один, називатимемо його `базовим`, який має містити всі синтаксичні конструкції, наявні у розробленій мові. Код у базовому прикладі має бути синтаксично та семантично коректним. Всі інші приклади містять код, який або демонструє більш складні конструкції (синтаксично чи семантично), або містять помилки різних типів.

2 Програмна реалізація транслятора

2.1 Загальні положення

На вході транслятора – результати лексичного аналізу, включно з ознакою успішності розбору `FSuccess` зі значеннями `(True,'Lexer')` або `(False,'Lexer')`, див. файл. `lex_my_lang_03.py`.

Постфіксний код програми вхідною мовою формується у списку `postfixCode`, до початку трансляції `postfixCode = []`.

Для трансляції використовується функція `postfixTranslator()`, яка викликає функцію-парсер `parseProgram()` тільки у випадку коректності лексичного аналізу:

```
def postfixTranslator():
    # чи був успішним лексичний розбір
    if (True,'Lexer') == FSuccess:
        return parseProgram()
```

Граматика вхідної мови, див. розд. 1.1, відрізняється від розглянутої у лабораторній роботі № 2, тим, що в ній нетермінал `Statement` не має альтернативи: `Statement = Assign`. Тому функції синтаксичного аналізатора `parseProgram()` та `parseStatementList()`, залишаться практично незмінними, а в `parseStatement()` залишиться тіьки виклик `parseAssign()` і не буде виклику `parseIf()`, див. файл.`postfixExpr_translator.py`:

```
def parseStatement():
    # print('\t\t parseStatement():')
    # прочитаємо поточну лексему в таблиці розбору
    numLine, lex, tok = getSymb()
    # якщо токен - ідентифікатор
    # обробити інструкцію присвоювання
    if tok == 'ident':
        parseAssign()
        return 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'):
        return False

else:
    # жодна з інструкцій не відповідає
    # поточній лексемі у таблиці розбору,
    failParse('невідповідність інструкцій',
                      (numLine,lex,tok,'ident або if'))
    return False
```

А от вже реалізація `parseAssign()` у складі транслятора відрізнятиметься від її реалізації у синтаксичному аналізаторі наявністю семантичних процедур.

2.2 Семантичні процедури

З наведеного на стор. 3 визначення зрозуміло, що ПОЛІЗ виразу можна знайти так:

- додати до ПОЛІЗУ вираз, якщо він – ідентифікатор, константа або оператор (після рекурсивної обробки аргументів (аргументу) опрератора);
- обробляючи вираз у дужках – ігнорувати дужки.

Доповнимо синтаксичний аналізатор, зважаючи на те, що він якраз обробляє всі елементи виразу рекурсивно, семантичними процедурами (у формі інструкцій).

Ці семантичні процедури, як ми тільки-но з'ясували, додаватимуть до ПОЛІЗУ вираз, якщо він – ідентифікатор, константа або оператор (після рекурсивної обробки аргументів опрератора), а при обробці виразу у дужках ігноруватимуть дужки.

Як це видно з граматики вхідної мови, див. розд. 1.1, всі ці випадки трапляються при обробці нетерміналів `Assign`, `Expression`, `Term` і `Factor`. Отож треба доповнити функції `parseAssign()`, `parseExpression()`, `parseTerm()` і `parseFactor()` семантичними процедурами.

Далі розглянемо приклад реалізації семантичних процедур у названих функціях.

2.2.1 У коді `parseAssign()`

Функція `parseAssign()`, див. файл `postfixExpr_translator.py`, фактично доповнена семантичними процедурами (діями) так:
- у рядку 11 – додавання ідентифікатора до `postfixCode`;
- у рядку 27 – додавання оператора `':='` до `postfixCode`, але тільки після завершення роботи функції `parseExpression()` у рядку 23. Про функцію `configToPrint()` див. розд 2.3.

Отже при обробці кожної інструкції присвоювання функція `parseAssign()` безпосередньо доповнює ПОЛІЗ двома символами у рядках 11 та 27. Все, що потрапить у ПОЛІЗ при розборі виразу `Expression`, додасть функція `parseExpression()` та викликані нею функції.

```
1  def parseAssign():
2      # номер запису таблиці розбору
3      global numRow, postfixCode
4      # print('\t'*4+'parseAssign():')
5
6      # взяти поточну лексему
7      # вже відомо, що це - ідентифікатор
8      _numLine, lex, _tok = getSymb()
9
10     # починаємо трансляцію інструкції присвоювання за означенням:
11     postfixCode.append(lex)      # Трансляція
12                                  # ПОЛІЗ ідентифікатора - ідентифікатор
13
14     if toView: configToPrint(lex,numRow)
15
16     # встановити номер нової поточної лексеми
17     numRow += 1
18
19     # print('\t'*5+'в рядку {0} - {1}'.format(numLine,(lex, tok)))
20     # якщо була прочитана лексема - ':='
21     if parseToken(':=','assign_op','\t\t\t\t\t'):
22         # розібрати арифметичний вираз
23         parseExpression()        # Трансляція (тут нічого не робити)
24                                  # ця функція сама згенерує
25                                  # та додасть ПОЛІЗ виразу
26
27         postfixCode.append(':=')# Трансляція
28                                  # Бінарний оператор  ':='
29                                  # додається після своїх операндів
30         if toView: configToPrint(':=',numRow)
31         return True
32     else: return False
```

2.2.2 У коді `parseExpression()`

Власне додавання коду до ПОЛІЗ здійснюється тільки у рядку 21, а саме додається оператор (лексема) `'+'` або `'-'` після того, як будуть оброблені його операнди у рядках 5 та 16:

```
1  def parseExpression():
2      global numRow, postfixCode
3      # print('\t'*5+'parseExpression():')
```

```
4       _numLine, lex, tok = getSymb()
5       parseTerm()                 # Трансляція (тут нічого не робити)
6                                   # ця функція сама згенерує
7                                   # та додасть ПОЛІЗ доданка
8       F = True
9       # продовжувати розбирати Доданки (Term)
10      # розділені лексемами '+' або '-'
11      while F:
12          _numLine, lex, tok = getSymb()
13          if tok in ('add_op'):
14              numRow += 1
15              # print('\t'*6+'в рядку {0} - {1}'.format(numLine,(lex, tok)))
16              parseTerm()         # Трансляція (тут нічого не робити)
17                                  # ця функція сама згенерує
18                                  # та додасть ПОЛІЗ доданка
19
20                                  # Трансляція
21              postfixCode.append(lex)
22                                  # lex - бінарний оператор  '+' чи '-'
23                                  # додається після своїх операндів
24              if toView: configToPrint(lex,numRow)
25          else:
26              F = False
27      return True
```

2.2.3 У `parseTerm()` та `parseFactor()`

Таким же чином додаються семантичні процедури у функціях розбору нетерміналів `Term` та `Factor` – `parseTerm()` та `parseFactor()` відповідно.

Код цих функцій містить інструкцію `postfixCode.append(lex)`, який доповнює ПОЛІЗ лексемою `lex`, що має симсл оператора (лексеми) `'*'` або `'/'` у функції `parseTerm()`, або константи чи ідентифікатора у функції `parseFactor()`, див. код функцій у файлі `postfixExpr_translator.py`.

2.3 Покроковий перегляд процесу трансляції

Для перегляду конфігурації транслятора була визначена функція `configToPrint(lex,numRow)`, виклик якої здійснюється командою `if toView: configToPrint(lex,numRow)` після кожної семантичної процедури, див. напр. рядки 14 та 30 коду функції `parseAssign()`. Функція `configToPrint(lex,numRow)` виводить у консоль: оброблену лексему, запис таблиці символів та отриманий після виконання семантичної процедури ПОЛІЗ:

```
def configToPrint(lex,numRow):
    stage = '\nКрок трансляції\n'
    stage += 'лексема: \'{0}\'\n'
    stage += 'tableOfSymb[{1}] = {2}\n'
```

```
    stage += 'postfixCode = {3}\n'
    # tpl = (lex,numRow,str(tableOfSymb[numRow]),str(postfixCode))
    print(stage.format(lex,numRow,str(tableOfSymb[numRow]),str(postfixCode)))
```

Переглянемо процес трансляції програми

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

Запуск транслятора та його робота виглядатимуть у консолі приблизно так:

```
>python postfixExpr_translator.py

Крок трансляції
лексема: 'v1'
tableOfSymb[2] = (3, 'v1', 'ident', 1)
postfixCode = ['v1']


Крок трансляції
лексема: '5.4'
tableOfSymb[5] = (3, '5.4', 'float', 1)
postfixCode = ['v1', '5.4']


Крок трансляції
лексема: '3'
tableOfSymb[7] = (3, '3', 'int', 2)
postfixCode = ['v1', '5.4', '3']


Крок трансляції
лексема: '+'
tableOfSymb[8] = (3, ')', 'par_op', '')
postfixCode = ['v1', '5.4', '3', '+']


Крок трансляції
лексема: 'a'
tableOfSymb[10] = (3, 'a', 'ident', 2)
postfixCode = ['v1', '5.4', '3', '+', 'a']
```

```
Крок трансляції
лексема: '/'
tableOfSymb[11] = (5, 'end', 'keyword', '')
postfixCode = ['v1', '5.4', '3', '+', 'a', '/']


Крок трансляції
лексема: ':='
tableOfSymb[11] = (5, 'end', 'keyword', '')
postfixCode = ['v1', '5.4', '3', '+', 'a', '/', ':=']

Translator: Переклад у ПОЛІЗ та синтаксичний аналіз
завершились успішно
```

3 Програмна реалізація інтерпретатора

3.1 Загальні положення

На вході інтерпретатора – побудований транслятором ПОЛІЗ вхідної програми у списку `postfixCode`, таблиці ідентифікаторів та констант, а також ознака успішності синтаксичного розбору та трансляції `FSuccess` зі значеннями `(True, 'Translator')` або `(False, 'Translator')`.

Постфіксний код, крім лексеми, містить і її токен, так для попереднього прикладу будується ПОЛІЗ у формі:

```
postfixCode = [('v1', 'ident'), ('5.4', 'float'), ('3', 'int'),
('+', 'add_op'), ('a', 'ident'), ('/', 'mult_op'), (':=', 'assign_op')]
```

Для цього замість семантичних процедур типу `postfixCode.append(lex)` використовуються інструкції `postfixCode.append((lex,tok))` див. файл `postfixExpr_translator_02.py` у теці `postfixExpr\postfixExpr_interpreter`.

Клас `Stack` імпортується з модуля `stack01`.

3.2 Інтерпретатор

Побудуємо інтерпретатор за алгоритмом зі стор. 4 з дотриманням семантики вхідної мови, див. розділ 1.2.4.

3.2.1 Базові кроки алгоритму

Функція `postfixInterpreter()`, за умови успішності етапу трансляції, викликає `postfixProcessing()`:

```
def postfixInterpreter():
    FSuccess = postfixTranslator()
    # чи була успішною трансляція
    if (True,'Translator') == FSuccess:
        print('\nПостфіксний код: \n{0}'.format(postfixCode))
        return postfixProcessing()
    else:
        # Повідомити про факт виявлення помилки
        print('Interpreter: Translator завершив роботу аварійно')
        return False
```

Оскільки ПОЛІЗ арифметичних виразів, як і інструкцій присвоювання, виконується послідовно у тому порядку, як він записаний у списку `postfixCode`, то функція `postfixProcessing()` просто за індексами елементів коду, див. рядок 5, вилучає найлівіший елемент, рядок 6, і якщо це – ідентифікатор, або константа, кладе його на стек, рядок 8, або передає на обробку, рядок 9, у `doIt(lex,tok)`:

```
1  def postfixProcessing():
2      global stack, postfixCode
3      maxNumb=len(postfixCode)
4      try:
5          for i in range(0,maxNumb):
6              lex,tok = postfixCode.pop(0)
7              if tok in ('int','float','ident'):
8                 stack.push((lex,tok))
9              else: doIt(lex,tok)
10             if toView: configToPrint(i+1,lex,tok,maxNumb)
11         return True
12     except SystemExit as e:
13         \# Повідомити про факт виявлення помилки
14         print('RunTime: Аварійне завершення програми з кодом {0}'.format(e))
15     return True
```

Зауважимо, що *вилучення запису* з postfix-коду запроваджено для зручності покрокового перегляду, і *можливе тільки* при відсутності у коді інструкцій умовного чи безумовного переходу, як у цій лабораторній роботі.

Оператори обробляються функцією `doIt(lex,tok)` таким чином. Якщо це – оператор присвоювання, то знімаються два операнди, рядки 5 і 7, і правий записують у таблицю ідентифікаторів як значення змінної з ідентифікатором-лівим операндом, рядки 15-16. Якщо це – це арифметичний оператор, то операнди та оператор передають на обробку функції
`processing_add_mult_op((lexL,tokL),lex,(lexR,tokR))`, рядок 26.

```
1  def doIt(lex,tok):
2      global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel
3      if (lex,tok) == (':=', 'assign_op'):
```

```
4           # зняти з вершини стека запис (правий операнд = число)
5           (lexL,tokL) = stack.pop()
6           # зняти з вершини стека ідентифікатор (лівий операнд)
7           (lexR,tokR) = stack.pop()
8
9           # виконати операцію:
10          # оновлюємо запис у таблиці ідентифікаторів
11          # ідентифікатор/змінна
12          # (index не змінюється,
13          # тип - як у константи,
14          # значення - як у константи)
15          tableOfId[lexR] = (tableOfId[lexR][0],
16                 tableOfConst[lexL][1], tableOfConst[lexL][2])
17      elif tok in ('add_op','mult_op'):
18          # зняти з вершини стека запис (правий операнд)
19          (lexR,tokR) = stack.pop()
20          # зняти з вершини стека запис (лівий операнд)
21          (lexL,tokL) = stack.pop()
22
23          if (tokL,tokR) in (('int','float'),('float','int')):
24              failRunTime('невідповідність типів',((lexL,tokL),lex,(lexR,tokR)))
25          else:
26              processing_add_mult_op((lexL,tokL),lex,(lexR,tokR))
27              # stack.push()
28              pass
29      return True
```

3.2.2 Семантика арифметики вхідної мови

Семантика арифметичних операторів мови, див. розд. 1.2.4, забезпечується функціями `processing_add_mult_op(operandL,operator,operandR)` та `getValue((valL,lexL,tokL),lex,(valR,lexR,tokR))`.

Функція `processing_add_mult_op((lexL,tokL),lex,(lexR,tokR))` визначає значення та типи значень лівого і правого операндів-ідентифікаторів, рядки 10 та 18 відповідно, чи значення констант, рядки 12 та 20. Потім передає всі ці лексеми, їх типи та значення функції `getValue((valL,lexL,tokL),lex,(valR,lexR,tokR))`, рядок 21, для обчислення результату. Помилка генерується, якщо один з операндів – ідентифікатор з невизначеним типом (`'type_undef'`) , р. 7 та 15.

```
1  def processing_add_mult_op(ltL,lex,ltR):
2      global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel
3      lexL,tokL = ltL
4      lexR,tokR = ltR
5      if tokL == 'ident':
6          if tableOfId[lexL][1] == 'type_undef':
7              failRunTime('неініціалізована змінна',
8                (lexL,tableOfId[lexL],(lexL,tokL),lex,(lexR,tokR)))
```

```
9           else:
10              valL,tokL = (tableOfId[lexL][2],tableOfId[lexL][1])
11      else:
12          valL = tableOfConst[lexL][2]
13      if tokR == 'ident':
14          if tableOfId[lexR][1] == 'type_undef':
15              failRunTime('неініціалізована змінна',
16                (lexR,tableOfId[lexR],(lexL,tokL),lex,(lexR,tokR)))
17          else:
18              valR,tokR = (tableOfId[lexR][2],tableOfId[lexR][1])
19      else:
20          valR = tableOfConst[lexR][2]
21      getValue((valL,lexL,tokL),lex,(valR,lexR,tokR))
```

Функція `getValue((valL,lexL,tokL),lex,(valR,lexR,tokR))` реалізує семантику обчислення, див. специфікацію у розд. 1.2.4: генерує винятки у випадку невідповідності типів аргументів, р. 5-6, або у випадку, якщо правий аргумент має значення нуль, р. 13-14. В решті випадків, в залежності від лексеми-оператора та типу операндів, засобами мови `python` обчислюється результат, р. 7-12 та 15-18, який потім кладеться на стек, р. 21, та заноситься до таблиці констант, р. 22.

```
1  def getValue(vtL,lex,vtR):
2      global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel
3      valL,lexL,tokL = vtL
4      valR,lexR,tokR = vtR
5      if (tokL,tokR) in (('int','float'),('float','int')):
6              failRunTime('невідповідність типів',((lexL,tokL),lex,(lexR,tokR)))
7      elif lex == '+':
8          value = valL + valR
9      elif lex == '-':
10         value = valL - valR
11     elif lex == '*':
12         value = valL * valR
13     elif lex == '/' and valR ==0:
14         failRunTime('ділення на нуль',((lexL,tokL),lex,(lexR,tokR)))
15     elif lex == '/' and tokL=='float':
16         value = valL / valR
17     elif lex == '/' and tokL=='int':
18         value = int(valL / valR)
19     else:
20         pass
21     stack.push((str(value),tokL))
22     toTableOfConst(value,tokL)
```

Функція `failRunTime()` обробляє помилки.

```
def failRunTime(str,tuple):
    if str == 'невідповідність типів':
        ((lexL,tokL),lex,(lexR,tokR))=tuple
        print('RunTime ERROR: \n\t Типи операндів відрізняються
                у {0} {1} {2}'.format((lexL,tokL),lex,(lexR,tokR)))
        exit(1)
    elif str == 'неініціалізована змінна':
        (lx,rec,(lexL,tokL),lex,(lexR,tokR))=tuple
        print('RunTime ERROR: \n\t Значення змінної {0}:{1} не визначене.
          Зустрілось у {2} {3} {4}'.format(lx,rec,(lexL,tokL),lex,(lexR,tokR)))
        exit(2)
    elif str == 'ділення на нуль':
        ((lexL,tokL),lex,(lexR,tokR))=tuple
        print('RunTime ERROR: \n\t Ділення на нуль
                у {0} {1} {2}. '.format((lexL,tokL),lex,(lexR,tokR)))
        exit(3)
```

3.3 Інтерпретація

Перевіримо інтерпретатор на прикладі програми

```
1 program
2 a := 5.4
3 b := 3.0
4 z := 2.0
5 v1 := (a + b)/z
6 end
```

```
>python postfix_Interpreter.py

------------------------------
Translator: Переклад у ПОЛІЗ та синтаксичний аналіз завершились успішно

Постфіксний код:
[('v1', 'ident'), ('5.4', 'float'), ('3.0', 'float'), ('+', 'add_op'), ('2.0', 'f
==============================
Interpreter run


 Таблиця символів
numRec     numLine    lexeme     token      index
1          1          program    keyword
2          3          v1         ident      1
3          3          :=         assign_op
4          3          (          par_op
5          3          5.4        float      1
```

```
6          3          +          add_op
7          3          3.0        float      2
8          3          )          par_op
9          3          /          mult_op
10         3          2.0        float      3
11         5          end        keyword

 Таблиця ідентифікаторів
Ident      Type            Value           Index
v1         type_undef      val_undef       1

 Таблиця констант
Const      Type       Value      Index
5.4        float      5.4        1
3.0        float      3.0        2
2.0        float      2.0        3

 Таблиця міток - порожня

Крок інтерпретації: 1
Лексема: ('v1', 'ident') у таблиці ідентифікаторів: v1:(1, 'type_undef', 'val_und
postfixCode=[('5.4', 'float'), ('3.0', 'float'), ('+', 'add_op'), ('2.0', 'float'
STACK:[ ('v1', 'ident')

Крок інтерпретації: 2
Лексема: ('5.4', 'float') у таблиці констант: 5.4:(1, 'float', 5.4)
postfixCode=[('3.0', 'float'), ('+', 'add_op'), ('2.0', 'float'), ('/', 'mult_op'
STACK:[ ('v1', 'ident') ('5.4', 'float')

Крок інтерпретації: 3
Лексема: ('3.0', 'float') у таблиці констант: 3.0:(2, 'float', 3.0)
postfixCode=[('+', 'add_op'), ('2.0', 'float'), ('/', 'mult_op'), (':=', 'assign_
STACK:[ ('v1', 'ident') ('5.4', 'float')        ('3.0', 'float')

Крок інтерпретації: 4
Лексема: ('+', 'add_op')
postfixCode=[('2.0', 'float'), ('/', 'mult_op'), (':=', 'assign_op')]
STACK:[ ('v1', 'ident') ('8.4', 'float')

Крок інтерпретації: 5
Лексема: ('2.0', 'float') у таблиці констант: 2.0:(3, 'float', 2.0)
postfixCode=[('/', 'mult_op'), (':=', 'assign_op')]
STACK:[ ('v1', 'ident') ('8.4', 'float')        ('2.0', 'float')
```

```
Крок інтерпретації: 6
Лексема: ('/', 'mult_op')
postfixCode=[(':=', 'assign_op')]
STACK:[ ('v1', 'ident') ('4.2', 'float')

Крок інтерпретації: 7
Лексема: (':=', 'assign_op')
postfixCode=[]
STACK:[

 Таблиця ідентифікаторів
Ident      Type            Value           Index
v1         float           4.2             1

 Таблиця констант
Const      Type       Value      Index
5.4        float      5.4        1
3.0        float      3.0        2
2.0        float      2.0        3
8.4        float      8.4        4
4.2        float      4.2        5

 Таблиця міток - порожня
```

Якщо у р. 4 вхідної програми замінити команду на `z := 2`, так що тип значення буде визначений як `int`, а решта лишиться `float`, то запуск інтерпретатора дасть результат:

```
...
Крок інтерпретації: 14
Лексема: ('z', 'ident') у таблиці ідентифікаторів: z:(3, 'int', 2)
postfixCode=[('/', 'mult_op'), (':=', 'assign_op')]
STACK:[ ('v1', 'ident') ('8.4', 'float')        ('z', 'ident')
RunTime ERROR:
         Типи операндів відрізняються у ('8.4', 'float') / ('z', 'int')
RunTime: Аварійне завершення програми з кодом 1
```

Натомість програма без конфлікту типів

```
1 program
2 a := 5
3 b := 3
4 z := 4
5 v1 := (a + b)/z
6 end
```

інтерпретується успішно з таблицею ідентифікаторів та констант:

```
 Таблиця ідентифікаторів
Ident      Type            Value           Index
a          int             5               1
b          int             3               2
z          int             4               3
v1         int             2               4

 Таблиця констант
Const      Type       Value      Index
5          int        5          1
3          int        3          2
4          int        4          3
8          int        8          4
2          int        2          5
```

Невизначена на момент використання змінна у арифметичному виразі, як-от `b` у програмі

```
1 program
2 a := 5
3
4 z := 4
5 v1 := (a + b)/z
6 end
```

спричинює повідомлення:

```
...
Крок інтерпретації: 9
Лексема: ('b', 'ident') у таблиці ідентифікаторів:
                         b:(4, 'type_undef', 'val_undef')
postfixCode=[('+', 'add_op'), ('z', 'ident'),
                        ('/', 'mult_op'), (':=', 'assign_op')]
STACK:[ ('v1', 'ident') ('a', 'ident')  ('b', 'ident')
RunTime ERROR:
         Значення змінної b:(4, 'type_undef', 'val_undef') не визначене.
                        Зустрілось у ('a', 'int') + ('b', 'ident')
RunTime: Аварійне завершення програми з кодом 2
```

4 Про звіт

4.1 Файли та тексти

Нагадую, що звіти про виконання лабораторних робіт треба надсилати у форматі pdf. Називати файли треба за шаблоном
`НомерГрупи.ЛР_Номер.ПрізвищеІніціали.pdf`, наприклад так
`ТВ-71.ЛР_3.АндрієнкоБВ.pdf`.

Тут *дуже важливо, щоб дефіс, підкреслювання та крапка були саме на своїх місцях, не було зайвих пробілів чи інших символів*. Одноманітність назв значно зменшує трудомісткість перевірки та імовірність помилки при обліку виконаних вами робіт.

Разом з *вільним розповсюдженням цих матеріалів* (поштою, месенджерами тощо), прошу *не розміщувати їх у вільному доступі*. Це позбавить мене зайвих клопотів при їх офіційному виданні (після доопрацювання), оскільки не треба буде пояснювати, що система контролю оригінальності тексту (т.зв. антиплагіату) знайшла попередню версію саме мого тексту, і що це не було його офіційним виданням.

Дякую за розуміння.

4.2 Форма та структура звіту

Вимоги до форми – мінімальні:

1. Прізвище та ім'я студента, номер групи, номер лабораторної роботи – у верньому колонтитулі. Нуменація сторінок – у нижньому колонтитулі. Титульний аркуш не потрібен.

2. На першому аркуші, угорі, – назва (тема) лабораторної роботи.

3. Далі, на першому ж аркуші та наступних – змістовна частина

Структура змістовної частини звіту:

1. Завдання саме Вашого (автора звіту) варіанту. Повне, включно з вимогами до арифметики (п'ять операцій, правоасоціативність піднесення до степеня) і т. і.

2. Граматика мови.

3. Специфікація інструкції присвоювання та арифметики розробленої мови (типи, константи, ідентифікатори, оператори тощо).

4. Про лексичний аналізатор коротко.

5. Про синтаксичний аналізатор коротко.

6. Формат таблиць: символів, ідентифікаторів, констант, міток (якщо використовуються).

7. Базовий приклад програми вхідною мовою для тестування транслятора та інтерпретатора.

8. Опис програмної реалізації транслятора та інтерпретатора.

9. План тестування, напр. як у Табл. 2.

10. Протокол тестування. Можна у формі скриншотів, текстових копій термінала тощо: фрагмент програми + результат обробки + ваш коментар та/чи оцінка результату

11. Висновки. Тут очікується власні оцінка/констатація/враження/зауваження автора звіту - виконавця лабораторної роботи.

План тестування може бути представлений у формі таблиці:

№	Тип випробування	Очікуванний результат	Кількість випробувань
8	базовий приклад	успішне виконання	1
9	помилка сканера або парсера	повідомлення від усіх наступних модулів	5
10	невизначений один з операндів	...	...
11	невідповідність типів операндів	...	...
12	ділення на нуль	...	...
13	вкладені конструкції	...	...
14	...	...	

Табл. 2: План тестування

Література

[1] Медведева В.М. Транслятори: внутрішнє подання програм та інтерпретація [Текст]: навч.посіб. [Текст]: навч.посіб. / В.М. Медведєва, В.А. Третяк/-К.: НТУУ «КПІ»,2015.-148с.