Матеріали до виконання лабораторної роботи № 5 Виконання ПОЛІЗ – програми

Юрій Стативка

Квітень, 2020 р.

Зміст

Вступ

Цей текст підготовлений з огляду на спричинену карантином відсутність лекційних занять як таких. Наводяться рекомендації для розширення інтерпретатора, побудованого в [2]. Програмний код розширення – НЕ додається.

Ті, хто вже зрозуміли як виконати це завдання чи вже виконали його, можуть тільки переглянути вимоги до оформлення звіту у розділі 2 .

Мета лабораторної роботи – розширення програмної реалізація інтерпретатора для виконання довільних програм розробленої вхідної мови програмування.

Постфікс-код

Програма у постфіксній нотації – послідовність ідентифікаторів змінних, констант, міток та операторів – арифметичних, присвоювання, умовного та безумовного переходу, `:` (`colon`), відношення (типу `>, <, =`), можливо логічних типу `and, or, not`, введення-виведення та інших, передбачених специфікацією вхідної мови.

Кожен елемент постфікс–коду зберігається у пам'яті за певною адресою, див. модельний приклад у Табл. 1.

3	5	a	+	<	m_1	JF	10	+	m_2	JMP	m_1	:	3	-	m_2	:
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Табл. 1: Приклад програми у постфіксній нотації

Алгоритм інтерпретації

В загальних рисах алгоритм інтерпретації програми у постфіксній формі, виглядає так:
Вхід: програма у постфіксній нотації.
Програма: послідовна (таблична) форма.
Дані: таблиця ідентифікаторів/змінних, констант, міток.
Вихід: значеня у таблиці ідентифікаторів/змінних та/або виведені повідомлення.

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, логічний, відношення тощо, то виконати операцію та покласти результат на стек;
   - якщо оператор – присвоювання, то занести значення правого операнда 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 у відповідності з семантикою оператора.

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. Специфікація розробленої вхідної мови (у т.ч. граматика).

2. Транслятор з розробленої вхідної мови.

3. Приклади програм для тестування інтерпретатора.

1 Розширення інтерпретатора

1.1 Загальні положення

На вході інтерпретатора – побудований транслятором, див. [3], ПОЛІЗ вхідної програми у списку `postfixCode`, таблиці ідентифікаторів, констант та міток, а також ознака успішності синтаксичного розбору та трансляції `FSuccess` зі значеннями `(True, 'Translator')` або `(False, 'Translator')`.

Розширюватимемо код інтерпретатора, побудованого в [2].

1.2 Інтерпретатор

Інтерпретатор має функціонувати за алгоритмом зі стор. 2 з дотриманням семантики вхідної мови.

1.2.1 Розширення базових функцій

Функція `postfixInterpreter()` трансформації не потребує. Інтерпретатор розглядає елемент постфікс-коду, номер якого зберігається у змінній `instrNum`.

Функція `postfixProcessing()` має бути трансформована, оскільки:

1. Порядок обробки інструкцій може змінюватись командами `JUMP` та `JF`. Тому номер наступного елемента постфікс-коду має обчислюватись після обробки поточного, рядки 12, 14, 17 та 19.

2. Розширився список токенів лексем, рядки 10 та 13.

3. Необхідно окремо обробити інструкції, які стосуються міток, рядки 13-14.

Для запобігання зациклюванню інтерпретатора при використанні `goto` передбачено змінну `cyclesNumb`, рядки 3, 6 та 7. Глобальний список `commandTrack` – для збирання інструкцій постфікс-коду при виконанні конкретної програми, рядок 9. Очевидно ці аспекти не є необхідними для інтерпретатора.

```
1  def postfixProcessing():
2      global stack, postfixCode, instrNum, commandTrack
3      cyclesNumb = 0
4      maxNumb=len(postfixCode)
5      try:
6          while instrNum < maxNumb and cyclesNumb < 100:
7              cyclesNumb += 1
8              lex,tok = postfixCode[instrNum]
9              commandTrack.append((instrNum,lex,tok))
10             if tok in ('int','float','ident','label','bool'):
```

```
11                  stack.push((lex,tok))
12                  nextInstr = instrNum + 1
13              elif tok in ('jump','jf','colon'):
14                  nextInstr = doJumps(tok)
15              else:
16                  doIt(lex,tok)
17                  nextInstr = instrNum + 1
18              if toView: configToPrint(instrNum,lex,tok,maxNumb)
19              instrNum = nextInstr
20          print('Загальна кількість кроків: {0}'.format(cyclesNumb))
21          return commandTrack
22      except SystemExit as e:
23          # Повідомити про факт виявлення помилки
24          print('RunTime: Аварійне завершення програми з кодом {0}'.format(e))
25      return commandTrack
```

Обробку інструкцій з токенами `jump`, `jf` і `colon` та обчислення номера наступної виконує `doJumps(tok)`, викликаючи відповідні функції:

```
1  def doJumps(tok):
2      if tok =='jump':
3          next = processing_JUMP()
4      elif tok =='colon':
5          next = processing_colon()
6      elif tok =='jf':
7          next = processing_JF()
8      return next
```

Функція `processing_JUMP()` знімає з вершини стека мітку та повертає значення цієї мітки як номер інструкції для виконання на наступному кроці.
Функція `processing_colon()` просто знімає з вершини стека непотрібну мітку та встановлює номер інструкції для наступного кроку на одиницю більшим за поточний номер.
Функція `processing_JF()` знімає з вершини стека мітку, потім знімає значення `BoolExpr` і, якщо це `false` – повертає значення мітки як номер інструкції для виконання на наступному кроці, інакше – номер, на одиницю більшим за поточний.

Зауваження: якщо перехід здійснюється не на `адресу мітки`, а на `адресу мітки + 2` (або так обчислюється значення мітки), то оператор `colon` і відповідна функція стають зайвими/непотрібними.

Оператори обробляються функцією `doIt(lex,tok)`:

```
1  def doIt(lex,tok):
2      global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel
3      # зняти з вершини стека правий операнд
4      (lexR,tokR) = stack.pop()
5      # зняти з вершини стека лівий операнд)
```

```
6       (lexL,tokL) = stack.pop()
7       if (lex,tok) == (':=', 'assign_op'):
8           # виконати операцію:
9           # оновлюємо запис у таблиці ідентифікаторів
10          # ідентифікатор/змінна  =  (index не змінюється,
11          #                           тип - як у константи,
12          #                           значення - як у константи)
13          tableOfId[lexL] = (tableOfId[lexL][0],  tableOfConst[lexR][1], \
14                                                  tableOfConst[lexR][2])
15      else:
16          processing_add_mult_rel_op((lexL,tokL),lex,(lexR,tokR))
17      return True
```

Відбувається це, у цій реалізації, так. Зі стека знімаються два операнди, рядки 4–6. Якщо агумент функції `(lex,tok)` – це оператор присвоювання, то значення (і токен) правого операнда записують у таблицю ідентифікаторів як значення (і токен) змінної з ідентифікатором – лівим операндом, рядки 7–14. Розширення необхідне у випадку, якщо оператор `(lex,tok)` – це оператор відношення (`tok=rel_op`), якого у попередній версії інтерпретатора не було. Тому для подальшої обробки викликається функція `processing_add_mult_rel_op((lexL,tokL),lex,(lexR,tokR))`, рядок 16, замість `processing_add_mult_op((lexL,tokL),lex,(lexR,tokR))` у попередній версії. Ідентифікатор функції тепер містить натяк (hint) на всі типи оброблюваних операторів. Втім змінити код необхідно, власне, тільки у функції `getValue(vtL,lex,vtR)`, яку вона викликає.

1.2.2 Семантика операторів вхідної мови

Семантика реалізованих тут бінарних операторів мови, як це зазначалось в [2], вимагає двох операндів одного і того ж типу. Це стосується і операторів з токеном `rel_op`.

Тому функцію `getValue(vtL,lex,vtR)` треба доповнити кодом, який:

1. Перевіряє, що операндии операторів `rel_op` – одного типу.
2. Обчислює результат застосування операторів типу `rel_op` до своїх операндів. Обчислює засобами мови, на якій написано інтерпретатор, тут це – `python`.

Результат застосування оператора `(>,rel_op)` у коді функції `getValue(vtL,lex,vtR)` може обчислюватись, наприклад, так:

```
elif lex == '>':
        value = str(valL > valR).lower()
```

Тобто, якщо оператор представлений лексемою `>`, а операнди мають значення `valL` та `valR`, то, засобами мови `python`:

1. Обчислити `valL > valR`. `python` поверне значення `True` або `False`.

2. Представити повернуте значення як рядок. `python` поверне значення `'True'` або `'False'`.

3. Привести повернуте значення до нижнього регістру. `python` поверне значення `'true'` або `'false'`.

4. Прийняти повернуте значення (`'true'` або `'false'`) як результат застосування оператора `>` до операндів `valL` та `valR`.

1.3 Інтерпретація

Перевіримо інтерпретатор на прикладі наведеної далі програми, очікуючи, що змінна з ідентифікатором `g` отримає значення 100, а змінні `s` та `f` залишаться невизначеними.

```
1  program
2  a := 1
3  b := 2
4  c :=3
5  if a >= b
6      then
7          if a <= c
8              then s := 5
9              else f := 500
10         endif
11     else g := 100
12 endif
13 end
```

```
>python postfixIF_interpreter.py
------------------------------
Translator: Переклад у ПОЛІЗ та синтаксичний аналіз завершились успішно

Постфіксний код:
[('a', 'ident'), ('1', 'int'), (':=', 'assign_op'), ('b', 'ident'),
 ('2', 'int'), (':=', 'assign_op'), ('c', 'ident'), ('3', 'int'),
 (':=', 'assign_op'), ('a', 'ident'), ('b', 'ident'), ('>=', 'rel_op'),
 ('m1', 'label'), ('JF', 'jf'), ('a', 'ident'), ('c', 'ident'),
 ('<=', 'rel_op'), ('m2', 'label'), ('JF', 'jf'),
 ('s', 'ident'), ('5', 'int'), (':=', 'assign_op'),
 ('m3', 'label'), ('JMP', 'jump'), ('m2', 'label'), (':', 'colon'),
 ('f', 'ident'), ('500', 'int'), (':=', 'assign_op'),
 ('m3', 'label'), (':', 'colon'), ('m4', 'label'), ('JMP', 'jump'),
 ('m1', 'label'), (':', 'colon'), ('g', 'ident'), ('100', 'int'),
 (':=', 'assign_op'), ('m4', 'label'), (':', 'colon')]
```

```
Загальна кількість кроків: 21
commandTrack = [(0, 'a', 'ident'), (1, '1', 'int'),
 (2, ':=', 'assign_op'), (3, 'b', 'ident'), (4, '2', 'int'),
 (5, ':=', 'assign_op'), (6, 'c', 'ident'), (7, '3', 'int'),
 (8, ':=', 'assign_op'), (9, 'a', 'ident'), (10, 'b', 'ident'),
 (11, '>=', 'rel_op'), (12, 'm1', 'label'), (13, 'JF', 'jf'),
 (33, 'm1', 'label'), (34, ':', 'colon'), (35, 'g', 'ident'),
 (36, '100', 'int'), (37, ':=', 'assign_op'),
 (38, 'm4', 'label'), (39, ':', 'colon')]

 Таблиця ідентифікаторів
Ident      Type            Value           Index
a          int             1               1
b          int             2               2
c          int             3               3
s          type_undef      val_undef       4
f          type_undef      val_undef       5
g          int             100             6

 Таблиця констант
Const      Type       Value      Index
1          int        1          1
2          int        2          2
3          int        3          3
5          int        5          4
500        int        500        5
100        int        100        6
false      int        false      7

 Таблиця міток
Label      Value
m1         33
m2         24
m3         29
m4         38
```

Замінивши у рядку 5 вхідної програми команду `if a >= b` на `if a < b`, очікуємо, що тепер змінна з ідентифікатором `s` отримає значення 5, а змінні `f` та `g` залишаться невизначеними. Результат інтерпредації:

```
...
 Таблиця ідентифікаторів
Ident      Type            Value           Index
a          int             1               1
b          int             2               2
c          int             3               3
s          int             5               4
f          type_undef      val_undef       5
g          type_undef      val_undef       6
...
```

Змінимо рядки 5–7 вхідної програми так, щоб перший логічний вираз мав значення `true`, а другий – `false`:

```
5 if a < b
6     then
7         if a = c
```

Тоді запуск інтерпретатора дасть очікуваний результат зі значенням 500 змінної `f` та невизначеними `s` та `g`:

```
...
 Таблиця ідентифікаторів
Ident      Type            Value           Index
a          int             1               1
b          int             2               2
c          int             3               3
s          type_undef      val_undef       4
f          int             500             5
g          type_undef      val_undef       6
...
```

2 Про звіт

2.1 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.

2.2 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. Специфікація усіх конструкцій, на які розширюється інтерпретатор (крім оператора присвоювання арифметичних значень та арифметики з лабораторної роботи № 3).
3. Про лексичний аналізатор коротко.
4. Про транслятор та синтаксичний аналізатор коротко.
5. Формат таблиць: символів, ідентифікаторів, констант, міток.
6. Базовий приклад програми вхідною мовою для тестування інтерпретатора.
7. Опис програмної реалізації інтерпретатора, незмінні з лабораторної роботи № 3 частини описувати не треба.
8. План тестування, напр. як у Табл. 2.
9. Протокол тестування. Можна у формі скриншотів, текстових копій термінала тощо: фрагмент програми + результат обробки + ваш коментар та/чи оцінка результату
10. 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
1	базовий приклад	успішне виконання, таблиці ідентифікаторів, констант, міток	1
2	вкладені конструкції	успішне виконання, таблиці ідентифікаторів, констант, міток	5
3	`if` з коректними/некоректними варіантами `BoolExpr`	успішне виконання або повідомлення про помилку + діагностика, таблиці ідентифікаторів, констант, міток	...
4	`if` з варіантами `ThenPart`	...	...
5	...	...	

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.

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

[3] Матеріали до виконання лабораторної роботи № 4 "Трансляція у ПОЛІЗ операторів розгалуження, циклу та введення-виведення"