Матеріали до виконання роботи № 4
Трансляція та виконання програм

Юрій Стативка

Листопад. 2023 р.

Зміст

Вступ

Тут розглядаються питання генерування проміжного коду, тобто трансляції програми з вхідної мови на її представлення засобами проміжної мови.

Існують різні типи проміжних мов — лінійне представлення (постфіксний запис — ПОЛІЗ), код віртуальної стекової машини (наприклад CLR чи JVM), триадресне представлення (наприклад четвірки, трійки), графічне представлення (синтаксичні дерева, орієнтовані ациклічні графи) тощо.

У цьому тексті розглядається побудова транслятора у постфікс-код (ПОЛІЗ). Матеріали про трансляцію для CLR та JVM (Common Language Runtime та Java Virtual Machine відповідно) будуть надані окремо.

Для виконання постфікс-коду використовується програма, яку будемо називати віртуальною постфікс-машиною (PSM).

Мета роботи — програмна реалізація транслятора (компілятора) у ПОЛІЗ, та код для CLR та JVM та виконання коду відповідними засобами (PSM, CLR та JVM).

Мотиваційний приклад

Розглянемо приклад програми, написаної певною вхідною мовою:

```
1  program
2  var
3     a, b, i:integer;
4     h, v1  :float;
5  begin
6      read(a)
7  m2:
8      read(b,h)
9      goto m1
10     h := 2
11 m1:
12     h := 1
13     if a < b
14        then v1 := (b - a)/b
15        else  v1 := (a - b)/b
16     endif
17     for i := a+1 step h+1 to (a+b)*2 do
18         h := h + v1
```

```
19          write(a+b,h,i)
20      endfor
21  end
```

Будемо вважати, що вона синтаксично і семантично коректна, зокрема у необхідних випадках виконується неявне приведення типів.

Очевидно, що список синтаксичних конструкцій, для яких має має бути згенерований цільовий код, містить, зокрема такі:

1. Інструкція (оператор) присвоювання.
2. Ідентифікатор (оператор) мітки з наступною двокрапкою як точка входження в потоці виконання, `m2:` та `m1:` в рядках 7 і 11 програми.
3. Інструкція (оператор) безумовного переходу з наступним ідентифікатором мітки, як `goto m1`у рядку 9.
4. Інструкція (оператор) розгалуження, як у рядках 13 — 10. Зауважимо, що її семантика потребує засобів оцінки значення логічного виразу та умовного переходу до `then` або `else` альтернативи.
5. Інструкція (оператор) циклу, як у рядках 17 –20.
6. Інструкції (оператори) введення та виведення значень одного чи списку елементів, як у рядках 6, 8 та 19.

Наведений приклад ілюструє певний невеликий набір конструкцій, які зустрічаються у мовах програмування, але транслятор створюється для конкретної мови у відповідності до її специфікації. Тож, якщо вхідна мова не містить оператора `goto` та міток, то і потреби транслювати такі конструкції немає.

Вхідна мова

Синтаксис

Нехай мова визначається граматикою з секцією оголошень з визначеними типами `int` і `float`:

```
Program       = program
                    DeclSection
                begin  StatementList  end

DeclSection   = var DeclarList

DeclarList    = {Ident '::' Type ';'}

Type          = int | float

StatementList = Statement  { Statement }
```

```
Statement = Assign
          | IfStatement
          | Read
          | Write

Assign = Ident ':=' Expression

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

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

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

IfStatement = if BoolExpr then Statement else Statement endif


BoolExpr = Expression ('='|'<='|'>='|'<'|'>'|'<>') Expression
         | true
         | false

Read = read '(' Ident ')'

Write = write '(' Ident ')'
```

Правила для нетерміналів, оброблені лексичним аналізатором, несуттєві для подальшого розгляду і тут не наводяться.

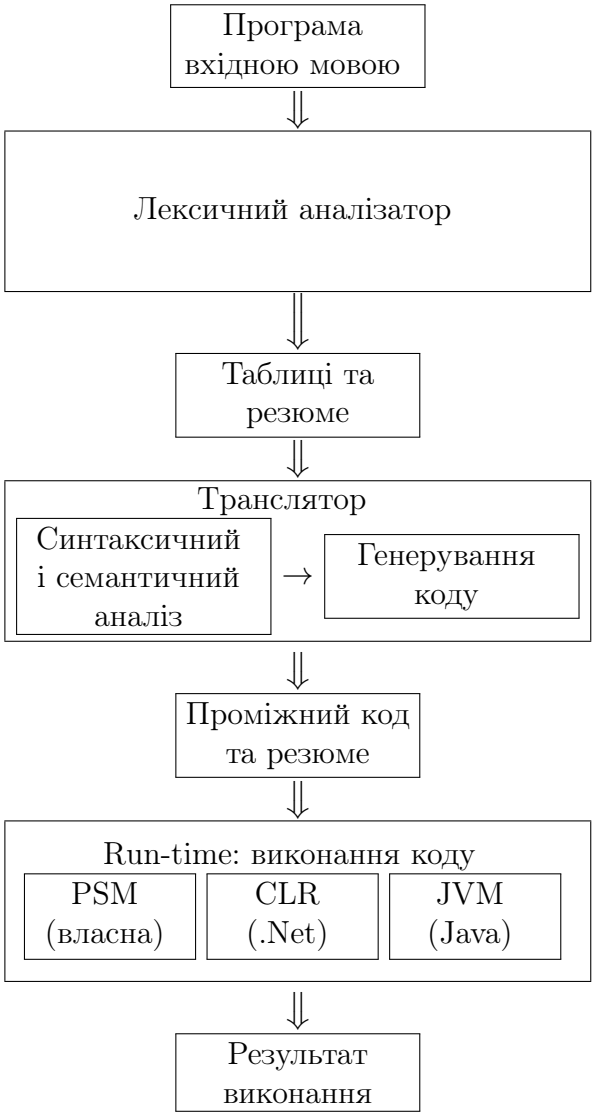
Зауважимо, що для кожної змінної передбачена окрема декларація, а кожна `if` - альтернатива містить тільки один оператор `(statement)`. У списках введення і виведення — тільки по одній змінній.

Семантика

У цій вхідній мові вважається, що операнди кожного бінарного оператора мають один і той же тип.

Загальна схема транслятора

Загальна схема взаємодії модулів транслятора та системи часу виконання показані на представленій тут схемі:



1 Постфіксна нотація

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. [2, розд. 1.1].

З прикладів, див. табл. 1, бачимо, що вираз у довільній (інфіксній, префіксній, постфіксній) нотації може бути записаний у постфіксній. Інструкція присвоювання, рядок № 9, теж вважається інфіксним виразом з оператором `:=` і перекладається за загальними правилами. У рядку № 5 унарний мінус теж

записується після свого аргументу, але щоб відрізняти його від (бінарного) оператора віднімання тут, він позначений символом `NEG`, а, наприклад в [2, розд. 1.1] — символом @. У випадку обчислення факторіала числа з постфіксним оператором `!`, переклад зводиться до переписування виразу, можливо, — на власний розсуд, з вибором іншого позначення оператора, як-от тут це — `FACT`[1].

№	Оператор	Тип конструкції	Конструкція	Постфіксна форма
1	Інфіксний	Вираз	a + b	a b +
2	Інфіксний	Вираз	5 - 7	5 7 -
3	Інфіксний	Вираз	a * 3	a 3 *
4	Інфіксний	Вираз	2 / 4	2 4 /
5	Префіксний	Вираз	-9	9 NEG
6	Префіксний	Інструкція	goto m1	m1 JMP
7	Постфіксний	Вираз	5!	5 FACT
8	Постфіксний	Інструкція	m1:	m1 :
9	Інфіксний	Інструкція	x := 1.1	x 1.1 :=

Табл. 1: Приклади перекладу простих синтаксичних конструкцій

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 [2, розд. 1.1.1].

[1] Звісно, що можливість обирати лексему для представлення оператора у проміжному коді є тільки у випадку, якщо розробник транслятора створює також і проміжну мову

Підхід, коли кожен символ програми, який не є елементом даних, вважається оператором з призначенням йому відповідних атрибутів, дуже докладно розглянуто у [2, розд. 1.1]. Проте існують і інші підходи, один з яких — рекурсивний, розглядається далі.

1.1 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 № 3).

1.2 Виконання ПОЛІЗ-програми

У цій роботі програма — це послідовність операторів присвоювання змінним значень арифметичних виразів. Вихід транслятора — програма, записана

у постфіксній формі — ПОЛІЗ-програма. ПОЛІЗ-програма може бути виконана засобами певної стекової машини.

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 значення 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 чи логічний, то виконати операцію та покласти результат на стек;
   - якщо оператор — присвоєння, 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 та покласти результат на стек.

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

2 Схема трансляції у ПОЛІЗ

Розглянемо можливі підходи до трансляції окремих синтаксичних конструкцій[2], враховуючи їх семантику та з метою з'ясування загальної схеми[3].

2.1 Оператор присвоювання

Правило `Assign = Ident ':=' Expression` для інструкції присвоювання передбачає, що спочатку до постфіксного коду додається ідентифікатор, потім — ПОЛІЗ виразу `Expression` і, нарешті, оператор (`:=`). До коду додаються лексеми разом з токенами, а ідентифікатор `Ident` — з позначкою `l-value`.

2.2 Мітка та оператор `colon`

Розпізнавання мітки може бути здійснене як на рівні лексичному, так і на синтаксичному за правилом, напр. `Мітка = ІдентМітки ':'` з відповідними семантичними процедурами. Проте бажано зберігати їх у таблиці розбору двома

[2]Пам'ятаємо, що [2] містить приклади трансляції великої кількості різноманітних синтаксичних конструкцій

[3]Про термінологію див. *Зауваження* на стор. 12

окремими рядками, а `ІдентМітки` заносити до таблиці міток, як це буде показано далі.

При трансляції конструкції `m1:`, якщо цільова мова містить мітки та команди переходу на них, вважаємо, що : — оператор, аргументом якого є мітка (ідентифікатор) `m1`. Зауважимо, що він — унарний постфіксний оператор, тобто, як і факторіал, він вже у постфіксній формі,див. Табл. 1, (тут ми не надавали оператору : нового позначення):

Семантичні процедури при трансляії оператора : такі: 1) додавання мітки до ПОЛІЗу (разом із токеном), 2) додавання до ПОЛІЗу оператора : (разом із токеном) і 3) оброблення таблиці міток (додати значення мітки, значення мітки — це номер, під яким вона зберігається перед двокрапкою у ПОЛІЗ-коді).

Наприклад, розбір та трансляція програми

```
program
    m5:
    m2:
end
```

дають такий результат:

```
 Таблиця символів
numRec     numLine    lexeme     token      index
1          1          program    keyword
2          2          m5         label
3          2          :          colon
4          3          m2         label
5          3          :          colon
6          4          end        keyword

 Таблиця ідентифікаторів
Ident      Type            Value           Index

 Таблиця констант
Const      Type       Value      Index

 Таблиця міток
Label      Value
m5         0
m2         2

Код програми у постфіксній формі (ПОЛІЗ):
[('m5', 'label'), (':', 'colon'), ('m2', 'label'), (':', 'colon')]
```

2.3 Оператор безумовного переходу `goto`

Унарний префіксний оператор у постфікс–коді теж записується після свого аргументу, як у Табл. 2:

№	Оператор	Вираз	Постфіксна форма
1	Префіксний	-9	9 NEG
2	Префіксний	goto m1	m1 JMP

Табл. 2: Приклади запису префіксних операторів

На рівні інструкцій ПОЛІЗу застосуємо лексему `JMP` з токеном `jump`. Розробник вільний у виборі відповідних лексем, так у [2] використовується `БП` — від **Б**езумовний **П**ерехід.Для нас важливо, що лексеми `JMP` та `goto` мають тотожну семантику, але у мовах проміжного та високого рівня відповідно, і при використанні не потребують додаткових уточнень.

Семантичні процедури при трансляції оператора `goto` зводяться до запису у таблицю розбору та внесення до ПОЛІЗ-коду.

У мовах з динамічною типізацією лексичний аналізатор, позбавлений зв'язку з парсером, може обробити ідентифікатор мітки як ідентифікатор та зробити відповідні записи у таблиці розбору та таблиці ідентифікаторів (а не таблиці міток). У такому випадку вказані недоліки мають бути скориговані на етапі синтаксичного розбору. Наступний приклад демонструє таку ситуацію для програми

```
1 program
2   m2:
3     goto m1
4     goto m2
5   m1:
6 end
```

```
 Таблиця символів
numRec     numLine    lexeme     token      index
1          1          program    keyword
2          2          m2         label
3          2          :          colon
4          3          goto       keyword
5          3          m1         ident      1
6          4          goto       keyword
7          4          m2         ident      2
8          5          m1         label
9          5          :          colon
10         6          end        keyword

 Таблиця ідентифікаторів
Ident      Type            Value           Index
m1         type_undef      val_undef       1
m2         type_undef      val_undef       2
```

```
 Таблиця констант
Const      Type       Value      Index

 Таблиця міток
Label      Value
m2         0
m1         6

Код програми у постфіксній формі (ПОЛІЗ):
[('m2', 'label'), (':', 'colon'), ('m1', 'ident'), ('JMP', 'jump'),
 ('m2', 'ident'), ('JMP', 'jump'), ('m1', 'label'), (':', 'colon')]
```

В такій ситуації, після лексичного аналізу, можна піти одним зі шляхів:

1. видалити з таблиці ідентифікаторів записи з лексемами ідентифікаторами-мітками; в таблиці розбору замінити записи типу `5:(3, 'm1', 'ident',1` на `5:(3, 'm1', 'label',''`.

2. Видалити з таблиці ідентифікаторів записи з лексемами ідентифікаторами-мітками; в таблиці розбору залишити записи як є, але при інтерпретації `('JMP','jump')` ігнорувати токен 'ident' лексеми на вершині стека.

3. Запропонувати інший спосіб.

2.4 Postfix-інструкція умовного переходу JF

Мови проміжного та низького рівня містять команди умовного переходу на мітку. Тут буде використовуватись постфіксна команда у формі
BoolExpr Label JF, яка здійснює перехід на мітку Label тільки у випадку, якщо BoolExpr має значення false. У формі таблиці фрагмент postfix-коду можна представити так (нижній рядок — адреси/номери елементів програми у постфіксній нотації): Якщо в результаті виконання інструкції <, номер 2, на

a	x	<	m_1	JF	...	...	m_1	:	...
0	1	2	3	4	5	...	n	n+1	n+2

стек буде покладено false, то команда JF здійснить перехід по мітці m_1, тобто наступною встановить команду з номером n. Інакше команда JF наступною встановить інструкцію з номером 5.

2.5 Оператор розгалуження if

Оператор розгалуження у граматиці представлений правилом для нетермінала IfStatement

```
IfStatement = if BoolExpression
                    then StatementList1
```

```
                else StatementList2
        endif
```

Для зручності графічного представлення перепишемо його з очевидними позначеннями (скороченими)

```
IfStatement = if BoolExpr then SL1 else SL2 endif
```

Для з'ясування способу трансляції у ПОЛІЗ розглянемо блок-схему оператора та розставимо мітки, наприклад так, як на Рис. 1. Семантика оператора очевидна:

1. Якщо `BoolExpr` має значення `False`, то перейти на мітку m_1, потім виконати `SL2`, потім перейти на m_2.
2. Якщо `BoolExpr` має значення `True`, то виконати `SL1`, потім перейти на m_2.

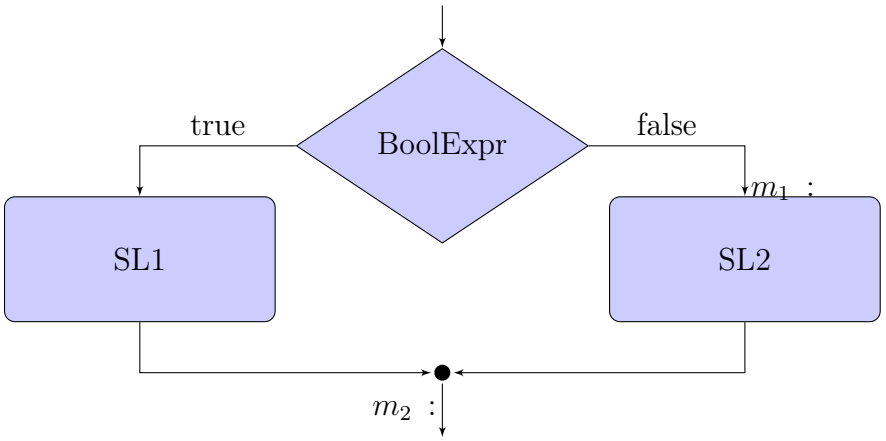


Рис. 1: Блок-схема оператора if

За допомогою ПОЛІЗ-інструкцій умовного та безумовного переходу схематично це можна представити, як у Табл. 3, де `BExpr` позначено `BoolExpr`, а квадратні дужки означають ПОЛІЗ виразу у дужках.

`[BExpr]`	m_1	`JF`	`[SL1]`	m_2	`JMP`	m_1	:	`[SL2]`	m_2	:	
n_1	n_1+1	n_1+2	n_1+3	n_2	n_2+1	n_2+2	n_2+3	n_2+4	n_3	n_3+1	

Табл. 3: Схема постфікс-коду оператора `if`

Отже, якщо `[BExpr]` має значення `false`, то, за інструкцією `JF`, перейти на мітку m_1, тобто до виконання інструкції з номером n_2+2, а, відтак, — до виконання `[SL2]`, а потім до мітки m_2.

Якщо ж `[BExpr]` має значення `true`, то інструкція `JF` не спрацьовує, далі виконується `[SL1]`, потім безумовний перехід `JMP`, інструкція n_2+1, передає управління на m_2, тобто інструкцію з номером n_3.

З порівняння структури оператора `if` вхідною та постфіксною мовами отримаємо схему трансляції, див. Рис. 2.

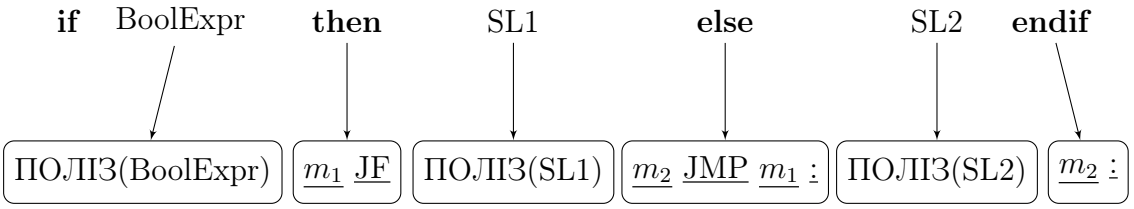


Рис. 2: Схема трансляції оператора `if`

Іншими словами, **схема трансляції** розглянутого тут оператора `if` полягає в таких діях:

1. Нічого не робити, коли зустрілась лексема `if`.
2. Виконати трансляцію у постфіксну форму логічного виразу `BoolExpr`.
3. Зустрівши лексему `then` — згенерувати нову мітку, хай це буде m_1; додати до ПОЛІЗу m_1 `JF`.
4. Виконати трансляцію у постфіксну форму списку операторів `SL1`.
5. Зустрівши лексему `else` — згенерувати нову мітку, хай це буде m_2; додати до ПОЛІЗу m_2 `JMP` m_1 `:`.
   Виконати семантичні процедури оператора `colon`, див. розділ 2.2.
6. Виконати трансляцію у постфіксну форму списку операторів `SL2`.
7. Зустрівши лексему `endif` — додати до ПОЛІЗу m_2 `:`.
   Виконати семантичні процедури оператора `colon`, див. розділ 2.2.

Зауваження. Схема трансляції (синтаксично керована) визначається в [1, розд. 2.3 та 5.4] як граматика з прикріпленими до продукцій (правил) програмних фрагментів. Самі фрагменти називаються також діями, а у випадку атрибутивних граматик — семантичними правилами.
В цьому тексті термін схема трансляції означає також графічне та/або словесне представлення процесу аналізу–синтезу, який забезпечує трансляцію з вхідної мови на цільову (ПОЛІЗ). Такі терміни як дія, семантична процедура, семантичне правило тощо вважаються тут синонімами.

2.6 Оператор циклу `for`

Розглянемо трансляцію оператора циклу такої структури:

```
for Prm := StartExpr step StepExpr to TargExpr do
   StatementList
endfor
```

Вважаючи позначення очевидними, розглянемо його семантику:

1. На першій ітерації параметру циклу `Prm` присвоїти значення виразу `StartExpr`.

2. Обчислити значення `StepExpr`. Але якщо це перша ітерація, то ігнорувати його, інакше збільшити параметр циклу `Prm` на значення кроку `StepExpr`, тобто `Prm := Prm + StartExpr`.

3. Перевірити, що значення параметра циклу знаходиться в інтервалі між `StartExpr` та `TargExpr`. Якщо це так, то виконати тіло циклу `StatementList` та перейти до обчислення `StepExpr`, п. 2, інакше вийти з циклу.

Позначивши ознаку першої/непершої ітерації як `r1`, значення 1 і 0 відповідно, та змінну для зберігання значення `StepExpr` як `r2`, побудуємо блок-схему, встановивши мітки для переходів, див. Рис. 3.

Тоді схема трансляції та відповідні семантичні процедури, можуть бути представлені, див. також Табл. 4, так:

№	Елемент	Переклад
1	`for`	
2	`Prm := StartExpr`	`Prm ПОЛІЗ(StartExpr) :=`
3	`step`	r_1 `1 :=` m_1 `:` r_2
4	`StepExpr`	`ПОЛІЗ(StepExpr)`
5	`to`	`:=` r_1 `0 =` m_2 `JF Prm Prm` r_2 `+ :=` m_2 `:` r_1 `0 :=Prm`
6	`TargExpr`	`ПОЛІЗ(TargExpr)`
7	`do`	`-` r_2 `* 0` $\leq$ m_3 `JF`
8	`StatementList`	`ПОЛІЗ(StatementList)`
9	`endfor`	m_1 `JMP` m_3 `:`

Табл. 4: Схема трансляції оператора `for`

1. Нічого не робити, коли зустрілась лексема `for`.

2. Додати до ПОЛІЗ-коду три елементи — `Prm ПОЛІЗ(StartExpr) :=`.

3. Зустрівши лексему `step` — згенерувати нову мітку, хай це буде m_1; створити, якщо вони ще не створені, дві внутрішні змінні з зарезервованими ідентифікаторами r_1 та r_2; додати до ПОЛІЗу інструкції r_1 `1 :=` m_1 `:` r_2.

4. Додати до ПОЛІЗ-коду `ПОЛІЗ(StepExpr)`.

5. Зустрівши лексему `to` — згенерувати нову мітку m_2; додати до ПОЛІЗ-коду інструкції `:=` r_1 `0 =` m_2 `JF Prm Prm` r_2 `+ :=` m_2 `:` r_1 `0 :=Prm`.

6. Додати до ПОЛІЗ-коду `ПОЛІЗ(TargetExpr)`.

7. Зустрівши лексему `do` — згенерувати нову мітку m_3; додати до ПОЛІЗ-коду інструкції `-` r_2 `*0` $\leq$ m_3 `JF`.

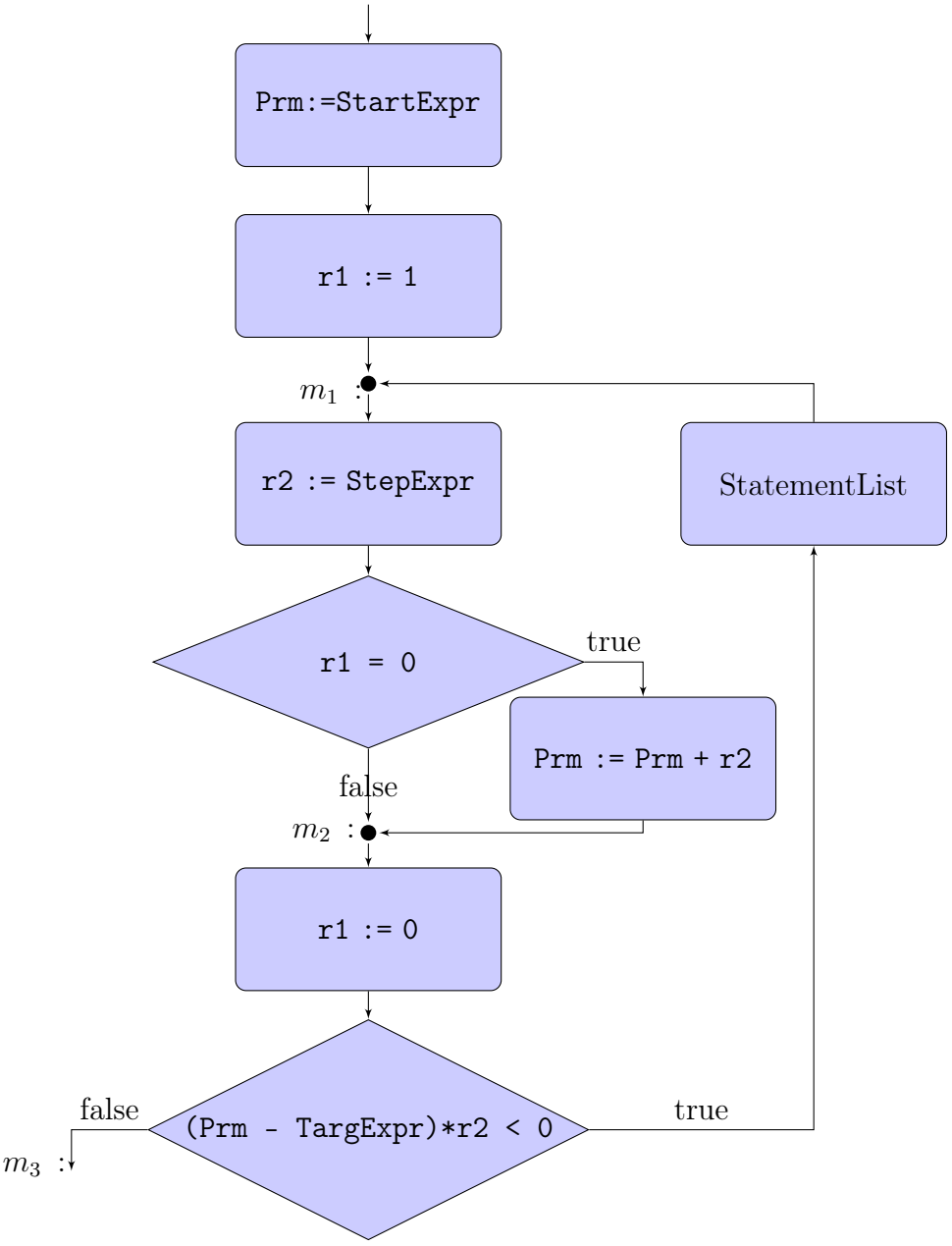


Рис. 3: Блок-схема інструкції for

8. Додати до ПОЛІЗ-коду `ПОЛІЗ(StatementList)`.
9. Зустрівши лексему `endfor` — додати до ПОЛІЗ-коду інструкції m_1 `JMP` m_3 `:`

Так, наприклад коду вхідною мовою

```
for i := a+1 step h/2 to (a+b)/2 do
    h := v1/i
    write(a+b,h,i)
endfor
```

відповідатиме постфіксний код:
`i a 1 + :=` r_1 `1 :=` m_1 `:` r_2 `h 2 / :=` r_1 `0 =` m_2 `JF i i` r_2 `+ :=` m_2 `:` r_1 `0 := i a b + 2 / - ` r_2 `* 0 ≤` m_3 `JF h v1 i / := a b + OUT h OUT i OUT` m_1 `JMP` m_3 `:`

Або у формі Табл. 5.

№	Елемент	Переклад
1	`for`	
2	`i := a+1`	`i a 1 + :=`
3	`step`	r_1 `1 :=` m_1 `:` r_2
4	`h / 2`	`h 2 /`
5	`to`	`:=` r_1 `0 =` m_2 `JF i i` r_2 `+ :=` m_2 `:` r_1 `0 := i`
6	`(a + b) / 2`	`a b + 2 /`
7	`do`	`-` r_2 `* 0 ≤` m_3 `JF`
8	`h := v1/i` `write(a+b,h,i)`	`h v1 i / :=` `a b + OUT h OUT i OUT`
9	`endfor`	m_1 `JMP` m_3 `:`

Табл. 5: Приклад трансляції оператора `for`

2.7 Оператори введення-виведення

Підхід до трансляції операторів введення-виведення розглянемо на прикладі нетермінала `Out`, див. також приклад у рядку 8 Табл. 5:

```
Out = write '(' OutExprList ')'
OutExprList = OutExpr  { ',' OutExpr }
```

Представимо правила для `Out` у формі:

```
Out = write '('  OutExpr  { ',' OutExpr } ')'
```

Схема трансляції `Out` може бути такою:

1. Нічого не робити, коли зустрілись лексеми `write` та `(`.

2. Зустрівши лексему `,` (кома) або `)` — додати до ПОЛІЗу інструкцію `OUT`, інакше — транслювати у ПОЛІЗ `OutExpr`.

Тут розглянуто випадок більш загальної інструкції `Out`, ніж `Write`, оголошена у складі граматики.

Аналогічно виконується трансляція інструкції введення, нетермінал `Read` граматики.

2.8 Загальна схема трансляції

Розглянуті приклади свідчать, що розробка схеми трансляції довільної синтаксичної конструкції вхідної мови складається з таких кроків:

1. Розглянути правила граматики.
2. Описати (специфікація мови) семантику конструкції.
3. Для складних конструкцій — побудувати блок-схему з використанням міток.
4. Описати блок-схему (у простих випадках — правила граматики) інструкціями постфікс-коду (з використанням міток та інструкцій умовного/безумовного переходу), записуючи `ПОЛІЗ(`X_i`)` замість високорівневих конструкцій X_i.
5. Із зіставлення правил граматики та опису з попереднього пункту встановити відповідність між високорівневими конструкціями X_i та `ПОЛІЗ(`X_i`)`, а решту фрагментів постфікс-коду асоціювати з терміналами правил граматики.
6. Додати до отриманої схеми опис інших необхідних дій (семантичних процедур), таких як, наприклад, додавання мітки до таблиці міток при її створенні.

3 Програмна реалізація транслятора

3.1 Загальні положення

Розроблений раніше синтаксичний і семантичний аналізатор тепер доповнимо компонентами перекладу (трансляції) синтаксичних конструкцій вхідної мови на мову цільову. У цьому тексті розглядається трансляція у постфікс-код (ПОЛІЗ).

Отже, завдання полягає у доповнені наявних програм компонентами для трансляції. Фактично, ці зміни стосуються функцій, що викликаються функцією `parseStatement()`

3.2 Код функцій

3.2.1 compileToPostfix(fileName)

Функція верхнього рівня `compileToPostfix(fileName)` викликає сканер для лексичного аналізу програми вхідною мовою з файлу `'fileName.my_lang'`, рядок 4, та викликає функцію синтаксичного розбору `parseProgram()`, рядок 12, доповнену компонентами для семантичного аналізу та генерування постфікс-коду.

```
1  def compileToPostfix(fileName):
2    global len_tableOfSymb , FSuccess
3    print('compileToPostfix: lexer Start Up\n')
4    FSuccess = lex(fileName)
5    print('compileToPostfix: lexer-FSuccess ={0}'.format(FSuccess))
6    # чи був успішним лексичний розбір
7    if (True,'Lexer') == FSuccess:
8      len_tableOfSymb = len(tableOfSymb)
9      print('-'*55)
10     print('compileToPostfix: Start Up compiler = parser + codeGenerator\n')
11     FSuccess = (False,'codeGeneration')
12     FSuccess = parseProgram()
13     if FSuccess == (True,'codeGeneration'):
14       serv()
15       savePostfixCode(fileName)
16   return FSuccess
```

Якщо всі перелічені фази компіляції здійснюються без помилок, рядки 7 і 13, то функція `serv()` виводить у консоль постфікс-код та таблиці, а функція `savePostfixCode(fileName)` зберігає програму у постфіксній нотації у файлі `'fileName.postfix'`, придатному для виконання засобами PSM.

Нехай файл `'test1.my_lang'` містить код

```
program
var
  a :: float;
  c :: int;
begin
 c := 1
 a:= 1.23
 if c > 0
   then a := 2.0 * a
   else a := 30.0 + a
 endif
end
```

Виклик функції `savePostfixCode('test1')` приводить до побудови постфікс-коду і виведення на консоль інформації:

```
compileToPostfix: lexer Start Up

compileToPostfix: lexer-FSuccess =(True, 'Lexer')
-----------------------------------------------------
compileToPostfix: Start Up compiler = parser + codeGenerator

Translator: Переклад у ПОЛІЗ та синтаксичний аналіз завершились успішно

 Таблиця міток
Label      Value
m1         18
m2         25

 Таблиця ідентифікаторів
Index    Ident    Type     Value
0        a        float    undefined
1        c        int      undefined

Код програми у постфіксній формі (ПОЛІЗ):

        №       postfixCode
        -       -----------
        0       ('c', 'l-val')
        1       ('1', 'int')
        2       (':=', 'assign_op')
        3       ('a', 'l-val')
        4       ('1.23', 'float')
        5       (':=', 'assign_op')
        6       ('c', 'r-val')
        7       ('0', 'int')
        8       ('>', 'rel_op')
        9       ('m1', 'label')
        10      ('JF', 'jf')
        11      ('a', 'l-val')
        12      ('2.0', 'float')
        13      ('a', 'r-val')
        14      ('*', 'mult_op')
        15      (':=', 'assign_op')
```

```
       16      ('m2', 'label')
       17      ('JMP', 'jump')
       18      ('m1', 'label')
       19      (':', 'colon')
       20      ('a', 'l-val')
       21      ('30.0', 'float')
       22      ('a', 'r-val')
       23      ('+', 'add_op')
       24      (':=', 'assign_op')
       25      ('m2', 'label')
       26      (':', 'colon')

postfix-код збережено у файлі test1.postfix
```

Постфікс-код зберігається у формі python-списку і може бути переглянутий у консолі командою `print(postfixCode)`

```
[('c', 'l-val'), ('1', 'int'), (':=', 'assign_op'), ('a', 'l-val'),
 ('1.23', 'float'), (':=', 'assign_op'), ('c', 'r-val'),('0', 'int'),
 ('>', 'rel_op'), ('m1', 'label'), ('JF', 'jf'), ('a', 'l-val'),
 ('2.0', 'float'), ('a', 'r-val'), ('*', 'mult_op'),
 (':=', 'assign_op'), ('m2', 'label'), ('JMP', 'jump'),
 ('m1', 'label'), (':', 'colon'), ('a', 'l-val'), ('30.0', 'float'),
 ('a', 'r-val'), ('+', 'add_op'), (':=', 'assign_op'),
 ('m2', 'label'), (':', 'colon')]
```

Збережений у файлі `test1.postfix` код ПОЛІЗ-програми містить всю необхідну для виконання у PSM інформацію:

```
.target: Postfix Machine
.version: 0.2

.vars(
   a     float
   c     int
)

.labels(
   m1    18
   m2    25
)

.constants(
   1     int
   1.23  float
   0     int
```

```
   2.0   float
   30.0  float
)

.code(
   c     l-val
   1     int
   :=    assign_op
   a     l-val
   1.23  float
   :=    assign_op
   c     r-val
   0     int
   >     rel_op
   m1    label
   JF    jf
   a     l-val
   2.0   float
   a     r-val
   *     mult_op
   :=    assign_op
   m2    label
   JMP   jump
   m1    label
   :     colon
   a     l-val
   30.0  float
   a     r-val
   +     add_op
   :=    assign_op
   m2    label
   :     colon
)
```

Методи класу PSM, визначеного у файлі 'postfixMachine.py', дозволяють завантажити та виконати програму за допомогою, наприклад, такого коду:

```
pm1=PSM()

#  завантаження .postfix - файлу
pm1.loadPostfixFile("test1")

pm1.postfixExec()
```

Виконання програми супроводжується виведенням у консоль протоколу:

```
postfixExec:
STACK:[
STACK:[ ('c', 'l-val')
STACK:[ ('c', 'l-val')  ('1', 'int')
-=-=-==========(:=,assign_op)  numInstr=2
STACK:[
STACK:[ ('a', 'l-val')
STACK:[ ('a', 'l-val')  ('1.23', 'float')
-=-=-==========(:=,assign_op)  numInstr=5
STACK:[
STACK:[ ('c', 'r-val')
STACK:[ ('c', 'r-val')  ('0', 'int')
-=-=-==========(>,rel_op)  numInstr=8
STACK:[ ('true', 'bool')
STACK:[ ('true', 'bool')        ('m1', 'label')
+=+=+==========(JF,jf)  numInstr=10 nextNumInstr=11
STACK:[
STACK:[ ('a', 'l-val')
STACK:[ ('a', 'l-val')  ('2.0', 'float')
STACK:[ ('a', 'l-val')  ('2.0', 'float')        ('a', 'r-val')
-=-=-==========(*,mult_op)  numInstr=14
STACK:[ ('a', 'l-val')  ('2.46', 'float')
-=-=-==========(:=,assign_op)  numInstr=15
STACK:[
STACK:[ ('m2', 'label')
+=+=+==========(JMP,jump)  numInstr=17 nextNumInstr=25
STACK:[
STACK:[ ('m2', 'label')
+=+=+==========(:,colon)  numInstr=26 nextNumInstr=27
STACK:[
```

Переглянути таблиці, постфікс-код та відповідність між номерами інструкцій у `pm1.postfixCode` та номерами відповідних рядків у файлі `test1.postfix` можна, наприклад, так:

```
print(f"pm1.tableOfId:\n  {pm1.tableOfId}\n")
print(f"pm1.tableOfLabel:\n  {pm1.tableOfLabel}\n")
print(f"pm1.tableOfConst:\n  {pm1.tableOfConst}\n")
print(f"pm1.postfixCode:\n  {pm1.postfixCode}\n")

for i in range(0,len(pm1.postfixCode)):
  s= "{0:4}  {1:4}   {2}".format(i, pm1.mapDebug[i], pm1.postfixCode[i])
  print(s)

print(f"pm1.mapDebug:\n  {pm1.mapDebug}\n")
```

```
pm1.tableOfId:
  {'a': (1, 'float', 2.46), 'c': (2, 'int', 1)}

pm1.tableOfLabel:
  {'m1': '18', 'm2': '25'}

pm1.tableOfConst:
  {'1': (1, 'int', 1), '1.23': (2, 'float', 1.23),
   '0': (3, 'int', 0), '2.0': (4, 'float', 2.0),
   '30.0': (5, 'float', 30.0)}

pm1.postfixCode:
  [('c', 'l-val'), ('1', 'int'), (':=', 'assign_op'), ('a', 'l-val'),
   ('1.23', 'float'), (':=', 'assign_op'), ('c', 'r-val'),
   ('0', 'int'), ('>', 'rel_op'), ('m1', 'label'), ('JF', 'jf'),
   ('a', 'l-val'), ('2.0', 'float'), ('a', 'r-val'), ('*', 'mult_op'),
   (':=', 'assign_op'), ('m2', 'label'), ('JMP', 'jump'),
   ('m1', 'label'), (':', 'colon'), ('a', 'l-val'), ('30.0', 'float'),
   ('a', 'r-val'), ('+', 'add_op'), (':=', 'assign_op'),
   ('m2', 'label'), (':', 'colon')]

   0    23   ('c', 'l-val')
   1    24   ('1', 'int')
   2    25   (':=', 'assign_op')
   3    26   ('a', 'l-val')
   4    27   ('1.23', 'float')
   5    28   (':=', 'assign_op')
   6    29   ('c', 'r-val')
   7    30   ('0', 'int')
   8    31   ('>', 'rel_op')
   9    32   ('m1', 'label')
  10    33   ('JF', 'jf')
  11    34   ('a', 'l-val')
  12    35   ('2.0', 'float')
  13    36   ('a', 'r-val')
  14    37   ('*', 'mult_op')
  15    38   (':=', 'assign_op')
  16    39   ('m2', 'label')
  17    40   ('JMP', 'jump')
  18    41   ('m1', 'label')
  19    42   (':', 'colon')
  20    43   ('a', 'l-val')
  21    44   ('30.0', 'float')
  22    45   ('a', 'r-val')
  23    46   ('+', 'add_op')
  24    47   (':=', 'assign_op')
  25    48   ('m2', 'label')
  26    49   (':', 'colon')
```

```
pm1.mapDebug:
  {0: 23, 1: 24, 2: 25, 3: 26, 4: 27, 5: 28, 6: 29, 7: 30, 8: 31,
   9: 32, 10: 33, 11: 34, 12: 35, 13: 36, 14: 37, 15: 38, 16: 39,
   17: 40, 18: 41, 19: 42, 20: 43, 21: 44, 22: 45, 23: 46, 24: 47,
   25: 48, 26: 49}
```

3.2.2 parseStatement()

Ця функція організована звичайним чином, містить виклики функцій для розбору усіх можливих інструкцій мови. Додатково, показано як могли би викликатись функції `parseGoto()` та `parseLabel()`, якби ця мова містила інструкцію `goto`.

```
1  def parseStatement():
2    # print('\t\t parseStatement():')
3    # прочитаємо поточну лексему в таблиці розбору
4    numLine, lex, tok = getSymb()
5    # якщо токен - ідентифікатор
6    # обробити інструкцію присвоювання
7    if tok == 'ident':
8      parseAssign()
9      return True
10   # якщо лексема - ключове слово 'if'
11   # обробити інструкцію розгалуження
12   elif (lex, tok) == ('if','keyword'):
13     parseIf()
14     return True
15   elif (lex, tok) == ('read','keyword'):
16     parseRead()
17     return True
18   elif (lex, tok) == ('write','keyword'):
19     parseWrite()
20     return True
21   elif tok == 'label':
22     parseLabel()                        # stub
23     return True
24   elif (lex, tok) == ('goto','keyword'):
25     parseGoto()                         # stub
26     return True
27   # тут - ознака того, що всі інструкції були коректно
28   # розібрані і була знайдена остання лексема програми.
29   # тому parseStatement() має завершити роботу
30   elif (lex, tok) == ('end','keyword'):
31     return False
32   else:
```

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

3.2.3 parseAssign()

Функція parseAssign(), фактично доповнена семантичними процедурами (діями) так:
- у рядку 15 — додавання ідентифікатора до postfixCode;
- у рядку 36 — додавання оператора ':=' до postfixCode, але тільки після завершення роботи функції parseExpression() у рядку 28.

Про функцію configToPrint() див. розд. 3.3.

Отже при обробці кожної інструкції присвоювання функція parseAssign() безпосередньо доповнює ПОЛІЗ двома записами у рядках 15 та 36. 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
4
5    # взяти поточну лексему
6    # вже відомо, що це - ідентифікатор
7    _numLine, lex, tok = getSymb()
8
9    lType = getTypeVar(lex)
10
11   # починаємо трансляцію інструкції присвоювання за означенням:
12   # Трансляція
13   # ПОЛІЗ ідентифікатора - ідентифікатор
14   # для PSM (Postfix Stack Machine)
15   postfixCodeGen('lval',(lex,tok))
16
17   # для CLR (Common Language Runtime, .Net)
18   # для JVM (Java Virtual Machine)
19
20   if toView: configToPrint(lex,numRow)
21
22   # встановити номер нової поточної лексеми
23   numRow += 1
24
25   # якщо була прочитана лексема - ':='
26   if parseToken(':=','assign_op','\t\t\t\t\t'):
27     # розібрати арифметичний вираз
28     rType = parseExpression() # Трансляція (тут нічого не робити)
29                               # ця функція сама згенерує
```

```
30                              # та додасть ПОЛІЗ виразу
31      # просто перевіряємо типи на тотожність
32      if lType == rType:
33
34        # Трансляція
35        # для PSM
36        postfixCodeGen(':=',(':=','assign_op'))
37        # Бінарний оператор  ':='
38        # додається після своїх операндів
39
40        # для CLR (Common Language Runtime, .Net)
41        # для JVM (Java Virtual Machine)
42      else:
43        failParse('невідповідність типів',(numRow,lex,lType,rType))
44        return 'type_error'
45      if toView: configToPrint(':=',numRow)
46    return 'void'
```

3.2.4 Генерувати постфікс-код `postfixCodeGen(case,toTran)`

Функція генерування коду

```
def postfixCodeGen(case,toTran):
  if   case == 'lval':
    lex,tok = toTran
    postfixCode.append((lex,'l-val'))
  elif case == 'rval':
    lex,tok = toTran
    postfixCode.append((lex,'r-val'))
  else:
    lex,tok = toTran
    postfixCode.append((lex,tok))
```

3.2.5 Код `parseExpression()`

Власне додавання коду до ПОЛІЗ здійснюється тільки у рядку 23, а саме додається оператор (лексема) `'+'` або `'-'` після того, як будуть оброблені його операнди у рядках 3 та 14:

```
1   def parseExpression():
2     global numRow, postfixCode
3     lType = parseTerm()         # Трансляція (тут нічого не робити)
4                                 # ця функція сама згенерує
5                                 # та додасть ПОЛІЗ доданка
6     resType = lType
7     F = True
8     # продовжувати розбирати Доданки (Term)
```

```
9   # розділені лексемами '+' або '-'
10  while F:
11    _numLine, lex, tok = getSymb()
12    if tok in ('add_op'):
13      numRow += 1
14      rType = parseTerm() # Трансляція (тут нічого не робити)
15                          # ця функція сама згенерує
16                          # та додасть ПОЛІЗ доданка
17      # просто перевіряємо типи на тотожність
18      if lType == rType:
19        resType = lType
20                          # Трансляція
21        # Для PSM
22        # postfixCode.append((lex,tok))
23        postfixCodeGen(lex,(lex,tok))
24                          # lex - бінарний оператор  '+' чи '-'
25                          # додається після своїх операндів
26        if toView: configToPrint(lex,numRow)
27      else:
28        resType = 'type_error'
29        failParse('невідповідність типів2',(numRow,lType,lex,rType))
30    else:
31      F = False
32  return resType
```

3.2.6 parseTerm() та parseFactor()

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

Наприклад, функція parseFactor() містить явне додавання постфікс-коду у рядках 15 та 25, та неявне — у рядку 30:

```
1  def parseFactor():
2    global numRow, postfixCode
3    numLine, lex, tok = getSymb()
4    numRow += 1
5    # перша альтернатива для Factor
6    # якщо лексема - це ідентифікатор (тоді це - rval)
7    if tok == 'ident':
8      typeVar  = getTypeVar(lex)
9      indexVar = getIndexVar(lex)
10     typeRes = typeVar
11     #          Трансляція
12     # ПОЛІЗ ідентифікатора - ідентифікатор
13     # Для PSM
14     # postfixCode.append((lex,tok))
15     postfixCodeGen('rval',(lex,'rval'))
```

```
16      if toView: configToPrint(lex,numRow)
17    # друга альтернативи для Factor
18    # якщо лексема - це константа
19    elif tok in ('int','float'):
20      typeRes = tok
21      #           Трансляція
22      # ПОЛІЗ константи - константа
23      # Для PSM
24      # postfixCode.append((lex,tok))
25      postfixCodeGen('const',(lex,tok))
26      if toView: configToPrint(lex,numRow)
27    # третя альтернатива для Factor
28    # якщо лексема - це ліва дужка
29    elif lex=='(':
30      typeRes = parseExpression() # Трансляція (тут нічого не робити)
31                                  # ця функція сама згенерує та додасть
                                    ↪ ПОЛІЗ множника
32      parseToken(')','par_op','\t'*7) # дужки у ПОЛІЗ НЕ додаємо
33    else:
34      failParse('невідповідність у
        ↪ Expression.Factor',(numLine,lex,tok,'rel_op, int, float,
        ↪ ident або \'(\' Expression \')\''))
35    return typeRes
```

3.3 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 29 та 45 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 += 'postfixCode = {3}\n'
  print(stage.format(lex,numRow,str(tableOfSymb[numRow]),str(postfixCode)))
```

Переглянемо процес трансляції програми

```
program

v1 := (5.4 + 3)/a

end
```

Встановивши глобальну змінну `toView = True`, отримаємо очевидний результат:

```
Крок трансляції
лексема: 'v1'
postfixCode = [('v1', 'l-val')]


Крок трансляції
лексема: '5.4'
postfixCode = [('v1', 'l-val'), ('5.4', 'float')]


Крок трансляції
лексема: '3.1'
postfixCode = [('v1', 'l-val'), ('5.4', 'float'), ('3.1', 'float')]


Крок трансляції
лексема: '+'
postfixCode = [('v1', 'l-val'), ('5.4', 'float'), ('3.1', 'float'),
 ('+', 'add_op')]


Крок трансляції
лексема: 'a'
postfixCode = [('v1', 'l-val'), ('5.4', 'float'), ('3.1', 'float'),
 ('+', 'add_op'), ('a', 'r-val')]


Крок трансляції
лексема: '/'
postfixCode = [('v1', 'l-val'), ('5.4', 'float'), ('3.1', 'float'),
 ('+', 'add_op'), ('a', 'r-val'), ('/', 'mult_op')]


Крок трансляції
лексема: ':='
postfixCode = [('v1', 'l-val'), ('5.4', 'float'), ('3.1', 'float'),
 ('+', 'add_op'), ('a', 'r-val'), ('/', 'mult_op'),
 (':=', 'assign_op')]
```

3.3.1 `parseIf()`

Тут реалізована схема трансляції, див. розділ 2.5. Зіставлення див. у Табл 6

```
# розбір інструкції розгалуження за правилом
# IfStatement = if BoolExpr then Statement else Statement endif
# функція названа parseIf() замість parseIfStatement()
def parseIf():
    global numRow
    _, lex, tok = getSymb()
    if lex=='if' and tok=='keyword':
        # 'if' нічого не додає до ПОЛІЗу    # Трансляція
        numRow += 1
        parseBoolExpr()                     # Трансляція
        parseToken('then','keyword','\t'*5)
        # Згенерувати мітку m1 = (lex,'label')
        m1 = createLabel()
        postfixCode.append(m1)  # Трансляція
        postfixCode.append(('JF','jf'))
                                # додали m1 JF

        parseStatement()        # Трансляція

        parseToken('else','keyword','\t'*5)
        # Згенерувати мітку m2 = (lex,'label')
        m2 = createLabel()
        postfixCode.append(m2)  # Трансляція
        postfixCode.append(('JMP','jump'))
        setValLabel(m1) # в табл. міток
        postfixCode.append(m1)
        postfixCode.append((':','colon'))
                                # додали m2 JMP m1 :
        parseStatement()        # Трансляція
        parseToken('endif','keyword','\t'*5)
        setValLabel(m2) # в табл. міток
        postfixCode.append(m2)  # Трансляція
        postfixCode.append((':','colon'))
                                # додали m2 JMP m1 :
        return True
    else: return False
```

3.3.2 `createLabel()`

Програмно згенеровані мітки отримують ідентифікатори `'m'+настВільнНомер`, рядки 3–4. Якщо мітки з таким ідентифікатором немає у таблиці міток — додаємо її до таблиці міток з невизначеним значенням, рядки 5–7, інакше виконання програми переривається з виведенням повідомлення `'Конфлікт міток'`, рядки 9–12. Зауважимо, що можна було би шукати справді `настВільнНомер` у циклі, не обмежуючись номером `len(tableOfLabel)+1`, — тоді конфлікт імен міток не виникав би.

№	Дія схеми трансляції	Рядки
1	Нічого не робити, коли зустрілась лексема `if`	7
2	Виконати трансляцію логічного виразу `BoolExpr` у постфіксну форму	10
3	Зустрівши лексему `then` — згенерувати нову мітку, хай це буде m_1; додати до ПОЛІЗу m_1 `JF`	12–16
4	Виконати трансляцію у постфіксну форму списку операторів `SL1`	18
5	Зустрівши лексему `else` — згенерувати нову мітку, хай це буде m_2; додати до ПОЛІЗу m_2 `JMP` m_1 `:`.Виконати семантичні процедури оператора `colon`, див. розділ 2.2	21–28
6	Виконати трансляцію у постфіксну форму списку операторів `SL2`	29
7	Зустрівши лексему `endif` — додати до ПОЛІЗу m_2 `:`. Виконати семантичні процедури оператора `colon`, див. розділ 2.2	31–34

Табл. 6: Відповідність коду схемі трансляції оператора `if`

```
1  def createLabel():
2      global tableOfLabel
3      nmb = len(tableOfLabel)+1
4      lexeme = "m"+str(nmb)
5      val = tableOfLabel.get(lexeme)
6      if val is None:
7          tableOfLabel[lexeme] = 'val_undef'
8          tok = 'label' # # #
9      else:
10         tok = 'Конфлікт міток'
11         print(tok)
12     exit(1003)
13      return (lexeme,tok)
```

3.3.3 `setValLabe()`

Тут встановлюється значення мітки, у відповідності до семантичних процедур оператора `colon`, див. розділ 2.2. У рядку 4 номер запису у постфікс-коді, куди буде записана мітка, встановлюється як значення цієї мітки .

```
1  def setValLabel(lbl):
2      global tableOfLabel
3      lex,_tok = lbl
4      tableOfLabel[lex] = len(postfixCode)
5      return True
```

3.3.4 parseBoolExpr()

Синтаксичний аналізатор/транслятор доповнений функцією для розбору та трансляції логічних виразів за правилом:

```
BoolExpr = true
         | false
         | Expression ('='|'<='|'>='|'<'|'>'|'<>') Expression
```

де `Expression` означає арифметичний вираз.

```
1  def parseBoolExpr():
2      global numRow
3      numLine, lex, tok = getSymb()
4      if lex == 'true' or lex == 'false':
5          numRow += 1
6          postfixCode.append((lex,tok))   # Трансляція
7          return True
8      else:
9          parseExpression()               # Трансляція
10         numLine, lex, tok = getSymb()
11         numRow += 1
12         parseExpression()               # Трансляція
13     if tok in ('rel_op'):
14         postfixCode.append((lex,tok))   # Трансляція
15         # print('\t'*5+'в рядку {0} - {1}'.format(numLine,(lex, tok)))
16     else:
17         failParse('mismatch in BoolExpr',(numLine,lex,tok,'relop'))
18     return True
```

3.4 Трансляція програм з розгалуженням

Переглянемо процес трансляції програми

```
1  program
2  var
3   a :: int;
4   b :: int;
5   c :: int;
6   d :: int;
7   s :: int;
8  begin
9  if a + b < c then s := 10 else d := 100 endif
10 end
```

Запуск транслятора приводить до результатів, які виглядають у консолі приблизно так:

```
compileToPostfix: lexer Start Up

compileToPostfix: lexer-FSuccess =(True, 'Lexer')
-----------------------------------------------------
compileToPostfix: Start Up compiler = parser + codeGenerator

Translator: Переклад у ПОЛІЗ та синтаксичний аналіз завершились успішно

 Таблиця міток
Label      Value
m1         12
m2         17

 Таблиця ідентифікаторів
Index    Ident    Type     Value
0        a        int      undefined
1        b        int      undefined
2        c        int      undefined
3        d        int      undefined
4        s        int      undefined

Код програми у постфіксній формі (ПОЛІЗ):

        №       postfixCode
        -       -----------
        0       ('a', 'r-val')
        1       ('b', 'r-val')
        2       ('+', 'add_op')
        3       ('c', 'r-val')
        4       ('<', 'rel_op')
        5       ('m1', 'label')
        6       ('JF', 'jf')
        7       ('s', 'l-val')
        8       ('10', 'int')
        9       (':=', 'assign_op')
        10      ('m2', 'label')
        11      ('JMP', 'jump')
        12      ('m1', 'label')
        13      (':', 'colon')
        14      ('d', 'l-val')
        15      ('100', 'int')
        16      (':=', 'assign_op')
        17      ('m2', 'label')
        18      (':', 'colon')

postfix-код збережено у файлі test1.postfix
```

Трансляція виконана правильно. Справді, всі змінні створені, але їх значення не визначені із-за відсутності відповідних команд у вхідній програмі.

Перехід за міткою m_1 здійснюється на інструкцію постфіксного коду з номером 12, тобто, врешті, до присвоювання значення 100 змінній `d`. Якщо ж інструкція `JF` не спрацьовує (тобто, якщо на момент її виконання на вершині стека –лексема `true`), то виконується наступний за `JF` код — присвоєння значення 10 змінній з ідентифікатором `s` та безумовний перехід на мітку m_2.

Виконаємо трансляцію програми з вкладеними операторами

```
program
var
 a :: int;
 b :: int;
 c :: int;
 d :: int;
 s :: int;
begin
if a + b < c
    then
        if a = b
            then s := 5
            else d := 500
        endif
    else d := 100
endif
end
```

Результати виглядають у консолі приблизно так:

```
compileToPostfix: lexer Start Up

compileToPostfix: lexer-FSuccess =(True, 'Lexer')
-------------------------------------------------------
compileToPostfix: Start Up compiler = parser + codeGenerator

Translator: Переклад у ПОЛІЗ та синтаксичний аналіз завершились успішно

 Таблиця міток
Label      Value
m1         26
m2         17
m3         22
m4         31
```

```
 Таблиця ідентифікаторів
Index    Ident    Type     Value
0        a        int      undefined
1        b        int      undefined
2        c        int      undefined
3        d        int      undefined
4        s        int      undefined

Код програми у постфіксній формі (ПОЛІЗ):

        №       postfixCode
        -       -----------
        0       ('a', 'r-val')
        1       ('b', 'r-val')
        2       ('+', 'add_op')
        3       ('c', 'r-val')
        4       ('<', 'rel_op')
        5       ('m1', 'label')
        6       ('JF', 'jf')
        7       ('a', 'r-val')
        8       ('b', 'r-val')
        9       ('=', 'rel_op')
        10      ('m2', 'label')
        11      ('JF', 'jf')
        12      ('s', 'l-val')
        13      ('5', 'int')
        14      (':=', 'assign_op')
        15      ('m3', 'label')
        16      ('JMP', 'jump')
        17      ('m2', 'label')
        18      (':', 'colon')
        19      ('d', 'l-val')
        20      ('500', 'int')
        21      (':=', 'assign_op')
        22      ('m3', 'label')
        23      (':', 'colon')
        24      ('m4', 'label')
        25      ('JMP', 'jump')
        26      ('m1', 'label')
        27      (':', 'colon')
        28      ('d', 'l-val')
        29      ('100', 'int')
        30      (':=', 'assign_op')
        31      ('m4', 'label')
        32      (':', 'colon')

postfix-код збережено у файлі test1.postfix
```

Література

[1] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman Compilers: Principles, Techniques, and Tools. Addison Wesley; 2nd edition, 2006. – 1040 p.

[2] Медведєва 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.