Матеріали до виконання лабораторної роботи № 2 Лексичний аналіз методом діаграми станів

Юрій Стативка

Березень, 2022 р.

Зміст

Вступ

Цей текст містить послідовну розповідь про побудову лексичного аналізатора мови програмування з використанням діаграми станів. Код розглянутого прикладу лексичного аналізатора додається у архіві.

Ті, хто вже зрозуміли як виконати це завдання чи вже виконали його, можуть тільки переглянути вимоги до оформлення звіту у розділі 4 .

Мета лабораторної роботи – програмна реалізація лексичного аналізатора заданої мови з використанням діаграми станів.

Лексичний аналізатор

Лексичний аналізатор (сканер, лексер) – аналізує програму, написану вхідною мовою, щодо допустимості наявних у програмі лексем. Кажуть, що лексичний аналізатор перетворює потік символів (characters) вхідного потоку на потік токенів вихідного потоку, див. напр. [1, 2].

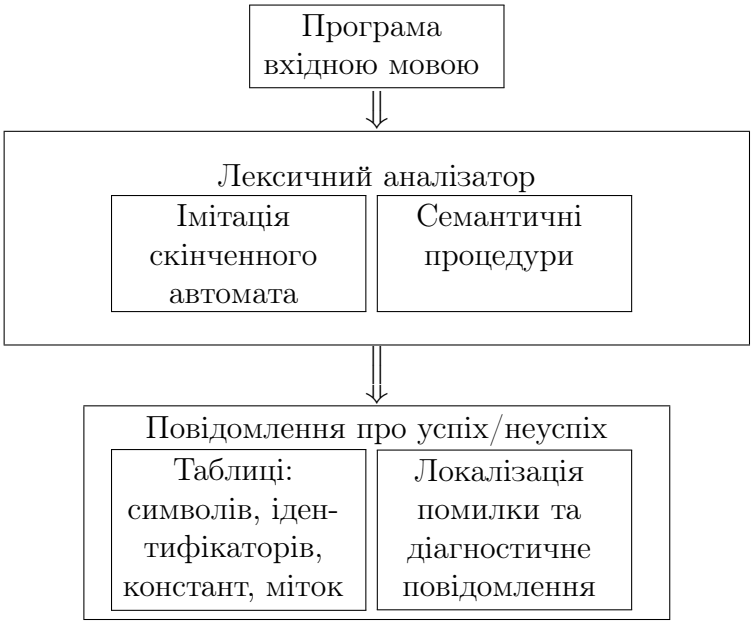
Вхідний потік будемо асоціювати з файлом, що містить програму вхідною мовою, а вихідний – з таблицею розбору програми (відому також як таблиця символів (table of symbol) програми), у якій послідовно зберігатимуться токени розпізнаних лексем. Деяку додаткову інформацію зберігатимемо у таблицях ідентифікаторів, констант та міток.

Отже, *Вхід* лексичного аналізатора – програма вхідною мовою.

Вихід, тобто результат роботи, лексичного аналізатора:

1. Повідомлення про успішність чи неуспішність лексичного аналізу.
   У випадку неуспішності:
   - локалізація помилки (номер рядка) у вхідній програмі;
   - діагностичне повідомлення.
2. Таблиця розбору *(таблиця символів програми)* – містить дані про кожну лексему програми: номер рядка, лексема, токен, індекс ідентифікатора або константи.
3. Таблиці ідентифікаторів, констант, міток (якщо потрібні).

Для розпізнавання лексем застосуємо програмний імітатор скінченного автомата (діаграми станів),



Діаграма станів

Скінченний автомат з деякою додатковою, корисною для програмування, інформацією називають діаграмою станів. Так на Рис. 1 функція переходів визначена не на окремих символах, а на класах символів (`Digit, Letter, other`), а мітка `*` нагадує про необхідність повернути вже прочитаний символ класу `other` у вхідний потік для наступного повторного зчитування та обробки.

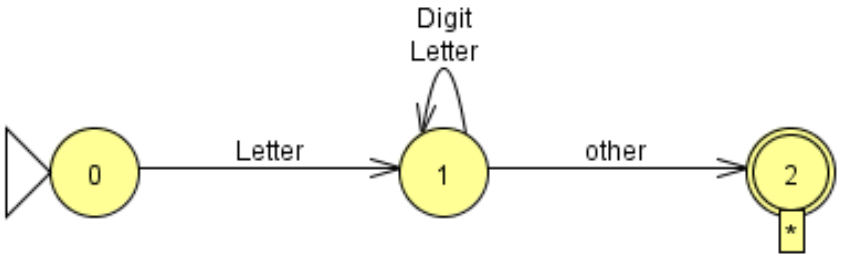


Рис. 1. Діаграма станів для розпізнавання ідентифікатора

План розробки лексичного аналізатора

Послідовність кроків для побудови лексичного аналізатора може бути такою:

1. Побудувати діаграму станів для розпізнавання лексем мови

   1) Визначити класи символів
   2) Побудувати графічно в JFLAP
   3) Записати у символьній формі, визначивши всі елементи кортежу $(Q, \Sigma, \delta, q_0, F)$

2. Визначити семантичні процедури для кожного заключного стану діаграми станів

3. Надати лексичному аналізаторові інформацію (про *лексеми мови = діаграму станів + таблицю лексем мови*)

   1) Визначити функцію (метод) для віднесення символів до одного з класів
   2) Обрати структури даних для зберігання інформації про діаграму станів, тобто про Q, Σ, δ, q_0 та F
   3) Обрати структуру даних для зберігання інформації про таблицю символів мови

4. Вихід лексичного аналізатора (результат роботи).

   1) Повідомлення про успішність/неуспішність лексичного аналізу вхідної програми, діагностичне повідомлення
   2) Таблиця розбору *(таблиця символів програми)*
   3) Таблиці ідентифікаторів, констант, міток (якщо потрібні)

5. Програмна реалізація сканера (лексичного аналізатора)

   1) Функції/методи для читання вхідного потоку та повернення у вхідний потік (*зірочка*)
   2) Функції/методи для переходу у наступний стан
   3) Функції/методи для реалізації семантичних процедур

6. Скласти план тестування та протестувати на:

   1) БАЗОВОМУ прикладі програми
   2) на прикладі з вкладеними конструкціями
   3) на прикладах, що демонструють виявлення помилок та діагностичні повідомлення

Необхідні програми та дані

Для виконання роботи потрібні:

1. Короткий опис мови (завдання).

2. Граматика мови.

3. Таблиця символів мови (таблиця лексем).
4. Програма JFLAP для побудови діаграми станів.
5. Базовий та інші приклади програм вхідною мовою для тестування.

Розглянутий далі приклад можна знайти у теці `my_lang_lexer`.

1 Початкові дані для розробки лексичного аналізатора

1.1 Короткий опис мови

Мова програмування `MicroLang1`.

Програма складається із списку інструкцій присвоювання між ключовими словами 'begin' та 'end'. Мова підтримує роботу з дійсними та цілими виразами. Дійсні константи представлені у форматі з плаваючою крапкою. Вирази можуть містити чотири основні (лівоасоціативні) арифметичні операції та круглі дужки.

1.2 Граматика мови

```
Program = program StatementList end

StatementList = Statement  { Statement }

Statement = Assign

Assign = Ident ':=' Expression

Ident = Letter {Letter | Digit }

Expression = Expression ('+' | '-' | '*' | '/') Expression
           | Id
           | Const
           | '(' Expression ')'

Const = Float | Int

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'
```

Всі нетермінали починаються з великої літери. Термінали – з маленької або у лапках.

1.3 Таблиця символів мови

Код	Приклади лексем	Токен	Неформальний опис
1	a, x1, z12f	id	ідентифікатор
2	123, 0, 521	int	ціле без знака
3	1.012, 34.76, 876.23	float	дійсне без знака
4	program	keyword	символ program
5	end	keyword	символ end
6	:=	assign_op	символ :=
7	+	add_op	символ +
8	-	add_op	символ -
9	*	mult_op	символ *
10	/	mult_op	символ /
11	(	par_op	символ (
12	)	par_op	символ)
13	.	dot	символ .
14	\t	ws	горизонтальна табуляція
15	\32	ws	пробіл
16	\n	eol	кінець рядка

Табл. 1. Таблиця лексем мови `MicroLang1`

2 Розробка лексичного аналізатора

2.1 Побудова діаграми станів

2.1.1 Класи символів

Оберемо класи символів: `Letter` та `Digit` для символів, позначених однойменними нетерміналами в граматиці, клас `dot` - для символу `.` (крапка), клас `ws` - для пробільних символів, клас `nl` - для символу нового рядка, клас `other` - для символів, що не належать до поточної лексеми.

2.1.2 Діаграма станів (графічне представлення)

Лексеми мови, представлені у Табл. 1 і такі, що містять більше одного символа, можуть бути розпізнані детермінованими автоматами, див. Рис. 2 – 5.

На Рис. 6 представлена узагальнена для мови `MicroLang1` діаграма станів, отримана шляхом об'єднання стартових станів часткових діаграм та доповнення кожного не заключного стану переходом за символом класу `other` у певний заключний стан з міткою `ERROR`. На діаграмі представлені також переходи для односимвольних лексем – пробільних символів, символу нового рядка, арифметичних операторів та дужок.

2.1.3 Діаграма станів (символьне представлення)

Діаграма станів як детермінований скінченний автомат:
$M = (Q, \Sigma, \delta, q_0, F)$

Множина станів:
$Q = \{0, 1, 2, 4, 5, 6, 9, 11, 12, 13, 14, 101, 102\}$

Алфавіт:
$\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., :, =, *, +, -, /, (,), \backslash n, \backslash t\}$

Функція переходів:
$\delta(0, Letter) = 1; \delta(1, Letter) = 1; \delta(1, Digit) = 1; \delta(1, other) = 2;$
$\delta(0, Digit) = 4; \delta(4, Digit) = 4; \delta(4, dot) = 5; \delta(5, Digit) = 5; \delta(5, other) = 6;$
$\delta(4, other) = 9;$
$\delta(0, `:') = 11; \delta(11, '=') = 12;$
$\delta(11, other) = 102;$
$\delta(0, ws) = 0;$
$\delta(0, nl) = 13;$
$\delta(0, `+') = 14; \delta(0, `-') = 14; \delta(0, `*') = 14; \delta(0, '/') = 14; \delta(0, `(') = 14; \delta(0, `)') = 14;$
$\delta(0, other) = 101$

Стартовий стан: $q_0 = 0$

Множина заключних станів: $F = \{2, 6, 9, 12, 13, 14, 101, 102\}$

Заключні стани, що потребують додаткової обробки:
$F_{star} = \{2, 6, 9\}$
$F_{ERROR} = \{101, 102\}$

2.2 Семантичні процедури

Перехід до заключного стану діаграми передбачає виконання певних дій, які знаходяться за межами формалізму скінченних автоматів і які називають семантичними процедурами. Перелік семантичних процедур для мови `MicroLang1` наведено у Табл. 2.

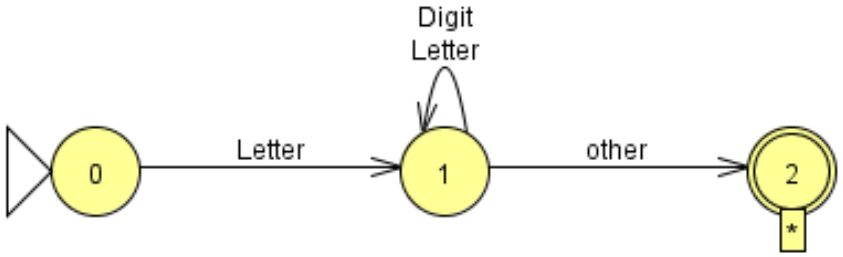


Рис. 2. Діаграма станів для розпізнавання ідентифікатора

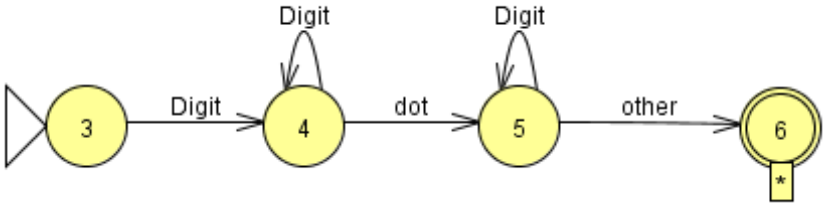


Рис. 3. Діаграма станів дійсної константи

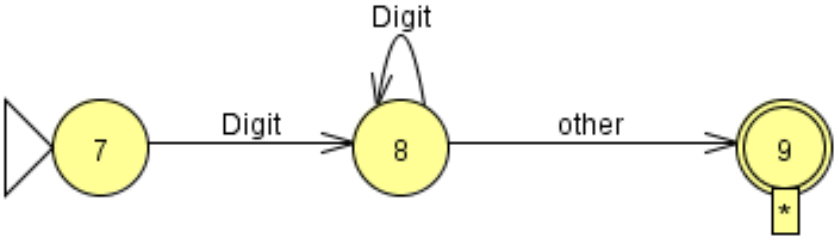


Рис. 4. Діаграма станів для цілої константи

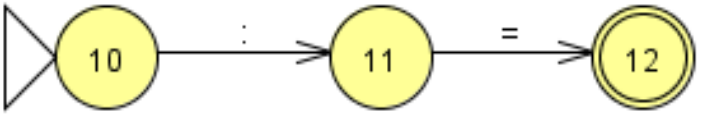


Рис. 5. Діаграма станів для розпізнавання оператора присвоювання

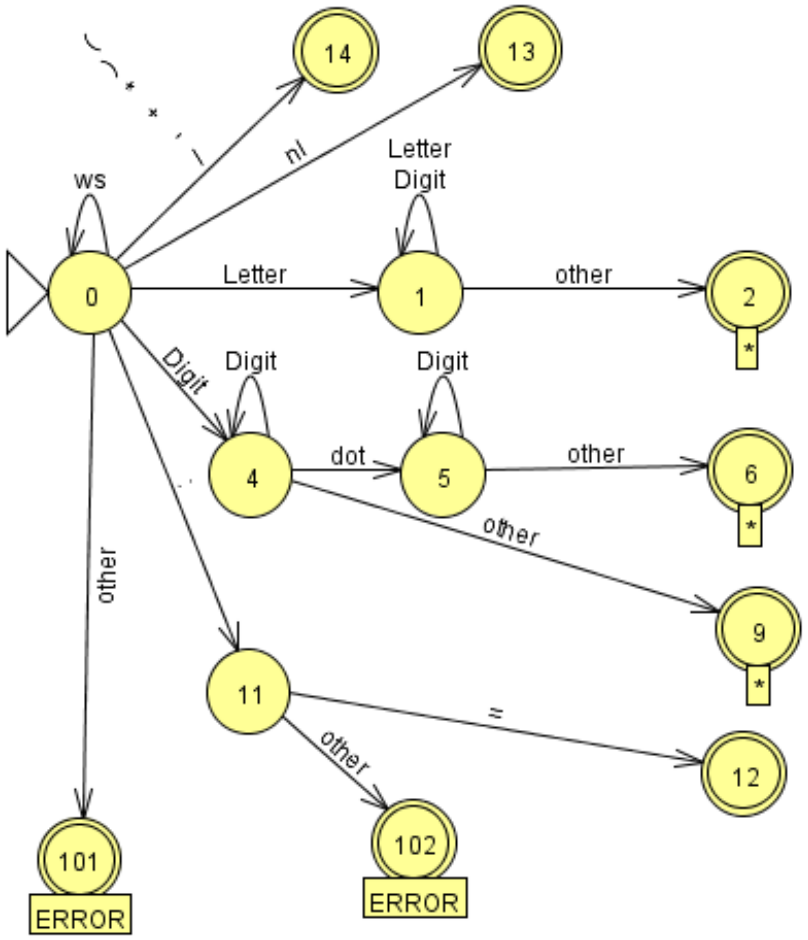


Рис. 6. Діаграма станів для розпізнавання довільних лексем мови `MicroLang1`

3 Програмна реалізація лексичного аналізатора

3.1 Діаграма станів і таблиця лексем у коді

Лексичний аналізатор реалізується мовою `python`. Усі необхідні таблиці реалізуються як словники (`dictionary`) мови `python`.

3.1.1 Визначення класу вхідного символа

Функція `classOfChar()` повертає клас, до якого належить символ `char`:

```
1 def classOfChar(char):
2      if char in '.' :
3           res="dot"
4      elif char in 'abcdefghijklmnopqrstuvwxyz' :
5           res="Letter"
6      elif char in "0123456789" :
7           res="Digit"
8      elif char in " \t" :
9           res="ws"
```

Стан	Токен	Семантичні процедури
2	`id` або `keyword`	Визначити: `id` чи `keyword`? Якщо `id` – обробити таблицю ідентифікаторів Занести лексему в таблицю розбору Повернути необроблений символ у вхідний потік Перейти у стартовий стан
6	`float`	Обробити таблицю констант Занести лексему в таблицю розбору Повернути необроблений символ у вхідний потік Перейти у стартовий стан
9	`int`	Обробити таблицю констант Занести лексему в таблицю розбору Повернути необроблений символ у вхідний потік Перейти у стартовий стан
12	`assign_op`	Занести лексему в таблицю розбору Перейти у стартовий стан
13	`nl`	Збільшити лічильник рядків на 1 Перейти у стартовий стан
14	`add_op, mult_op, par_op`	Занести лексему в таблицю розбору Перейти у стартовий стан

Табл. 2. Семантичні процедури

```
10      elif char in "\n" :
11           res="nl"
12      elif char in "+-:=*/()" :
13           res=char
14      else: res='символ не належить алфавіту'
15      return res
```

3.1.2 Діаграма станів у коді

Функція переходів δ представлена у формі словника (dictionary) stf (state-transition function) мови python:

```
1  # state-transition function
2  stf={(0,'Letter'):1,  (1,'Letter'):1, (1,'Digit'):1, (1,'other'):2,\
3       (0,'Digit'):4, (4,'Digit'):4, (4,'dot'):5,  (5,'Digit'):5, (5,'other'):6, \
4                                              (4,'other'):9, \
5       (0, ':'):11, (11,'='):12,\
6                         (11,'other'):102,\
7       (0, 'ws'):0, \
8       (0, 'nl'):13, \
9       (0, '+'):14, (0, '-'):14, (0, '*'):14, \
10      (0, '/'):14,  (0, '('):14, (0, ')'):14, \
11      (0, 'other'):101
12 }
```

```
13
14
15 initState = 0   # q0 - стартовий стан
16 F={2,6,9,12,13,14,101,102} # множина заключних станів
17 Fstar={2,6,9}   # зірочка
18 Ferror={101,102}# обробка помилок
```

3.1.3 Таблиця символів мови у коді

Таблиця символів мови реалізована у формі двох словників – `tableOfLanguageTokens` та `tableIdentFloatInt`. Перший містить інформацію про токени, що однозначно представлені переліком лексем, другий – про ідентифікатори та константи.

```
1 # Таблиця лексем мови
2 tableOfLanguageTokens = {'program':'keyword', 'end':'keyword',
3       ':=':'assign_op', '.':'dot', ' ':'ws', '\t':'ws', '\n':'nl',
4        '-':'add_op', '+':'add_op', '*':'mult_op',
5        '/':'mult_op', '(':'par_op', ')':'par_op'}
6 # Решту токенів визначаємо не за лексемою, а за заключним станом
7 tableIdentFloatInt = {2:'ident', 6:'float', 9:'int'}
```

3.2 Вихід лексичного аналізатора

3.2.1 Резюме про успішність лексичного розбору

Повідомлення формуються наступними командами:

1. про успішність (функція lex())
   – print('Lexer: Лексичний аналіз завершено успішно')
2. про неуспішність (функція lex())
   – print('Lexer: Аварійне завершення програми з кодом 0'.format(e))
3. діагностичні повідомлення у станах 101 та 102 відповідно (функція fail())
   – print('Lexer: у рядку ',numLine,' неочікуваний символ '+char)
   – print('Lexer: у рядку ',numLine,' очікувався символ =, а не '+char)

3.2.2 Таблиця символів програми (таблиця розбору)

Таблиця розбору реалізована як словник `tableOfSymb` у форматі:

```
{ n_rec : (num_line, lexeme, token, idxIdConst) }
```

де:
`n_rec` – номер запису в таблиці символів програми;
`num_line` – номер рядка вхідної програми;
`lexeme` – лексема;
`token` – токен лексеми;
`idxIdConst` – індекс ідентифікатора або константи у таблиці ідентифікаторів та констант відповідно; для інших лексем – порожній рядок.

3.2.3 Таблиця ідентифікаторів

Таблиця ідентифікаторів `tableOfId`:

```
{ Id : idxId) }
```

де:
`Id` – ідентифікатор (лексема);
`idxId` – індекс ідентифікатора у таблиці ідентифікаторів.

3.2.4 Таблиця констант

Таблиця констант `tableOfConst`:

```
{ Const : idxConst}
```

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

3.2.5 Таблиця міток

Таблиця міток `tableOfLabel`:

```
{ Label : idxLbl }
```

де:
`Label` –мітка (лексема);
`idxLbl` – індекс мітки у таблиці міток.

3.3 Програмна реалізація основних функцій

3.3.1 Функція верхнього рівня

Функція `lex()`, у циклі, див. рядки 11-21 коду, намагається проаналізувати всі символи програми вхідною мовою (`while numChar<lenCode`, рядок 11). Зчитування символу, віднесення його до одного з прийнятих класів та перехід у наступний стан здійснюється у рядках 12 - 14 коду. Коли перехід здійснюється у заключний, стартовий чи інший стан (див. рядки 15, 17 та 19), здійснюється, відповідно, виконання семантичних процедур (функція `processing()`), підготовка до пошуку нової лексеми (рядок 18), або приєднання прочитаного символу до поточної лексеми (рядок 20).

Якщо при розборі буде досягнуто заключного стану із множини F_{ERROR}, то фукнцією `fail()` буде згенеровано виняток, який буде перехоплено та оброблено у блоці `except`, рядки 15 - 18.

```
1 def lex():
2    global state,numLine,char,lexeme,numChar,FSuccess
3    try:
4       while numChar<lenCode:
```

```
5           char=nextChar()                 # прочитати наступний символ
6           classCh=classOfChar(char)       # до якого класу належить
7           state=nextState(state,classCh) # обчислити наступний стан
8           if (is_final(state)):           # якщо стан заключний
9              processing()                 #    виконати семантичні процедури
10          elif state==initState:          # якщо стан НЕ закл., а стартовий, то
11             lexeme=''                    #      збиратимемо нову лексему
12          else:                           # якщо стан НЕ закл. і не стартовий, то
13             lexeme+=char                 #      додати символ до лексеми
14       print('Lexer: Лексичний аналіз завершено успішно')
15    except SystemExit as e:
16       FSuccess = (False,'Lexer')        # Встановити ознаку неуспішності
17       # Повідомити про неуспіх
18       print('Lexer: Аварійне завершення програми з кодом {0}'.format(e))
```

3.3.2 Функції читання символу та його повернення у вхідний потік (*зірочка*)

Вхідна програма зберігається у рядку (String) sourceCode. Тому читання наступного символа реалізується функцією `nextChar()` шляхом зчитування символа з попередньо інкрементованим його номером `numChar`.

```
def nextChar():
   global numChar
   numChar+=1
   return sourceCode[numChar]
```

Повернення усимволу у вхідний потік реалізується функцією `putCharBack(numChar)` шляхом декрементації номера символу `numChar`.

```
def putCharBack(numChar):
   return numChar-1
```

3.3.3 Переходи у діаграмі станів

Функція `nextState(state,classCh)` намагається знайти значення у словнику `stf` значення за ключем `(state,classCh)`. Якщо таке значення знайдено, то воно і є ім'ям наступного стану, якщо ж запису з таким ключем немає – то перхоплюється згенерований системою `python` виняток `KeyError`, а перехід зі стану `state` здійснюється по класу символів `other`.

```
def nextState(state,classCh):
   try:
      return stf[(state,classCh)]
   except KeyError:
      return stf[(state,'other')]
```

3.3.4 Реалізація семантичних процедур

```
def processing():
   global state,lexeme,char,numLine,numChar, tableOfSymb
   if state==13: # \n
      numLine+=1
      state=initState
   if state in (2,6,9): # keyword, ident, float, int
      token=getToken(state,lexeme)
      if token != 'keyword':     # НЕ keyword
         index=indexIdConst(state,lexeme)
         print('{0:<3d} {1:<10s} {2:<10s} {3:<2d} '.format(numLine,\
                                                  lexeme,token,index))
         tableOfSymb[len(tableOfSymb)+1] = (numLine,lexeme,token,index)
      else:                      # якщо keyword
         print('{0:<3d} {1:<10s} {2:<10s} '.format(numLine,lexeme,token))
         tableOfSymb[len(tableOfSymb)+1] = (numLine,lexeme,token,'')
      lexeme=''
      numChar=putCharBack(numChar) # зірочка
      state=initState
   if state in (12,14): #12:         # assign_op # in (12,14):
      lexeme+=char
      token=getToken(state,lexeme)
      print('{0:<3d} {1:<10s} {2:<10s} '.format(numLine,lexeme,token))
      tableOfSymb[len(tableOfSymb)+1] = (numLine,lexeme,token,'')
      lexeme=''
      state=initState
   if state in Ferror:  #(101,102):  # ERROR
      fail()
```

4 Про звіт

4.1 Файли та тексти

Нагадую, що звіти про виконання лабораторних робіт треба створювати у Google Docs.

Називати файли треба за шаблоном `НомерГрупи.ЛР_Номер.ПрізвищеІніціали`, наприклад так `ТВ-з91.ЛР_1.АндрієнкоБВ`.

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.

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, див. заголовок цього документу.

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 (цілі, дійсні, п'ять операцій, правоасоціативність піднесення до степеня) і т. і.

2. Граматика мови.

3. Таблиця лексем.

4. Далі відповідно до плану, див. стор. 2.

5. На відміну від цього тексту, у звіті не потрібні проміжні результати (наприклад часткові діаграми) та роз'яснення загальних речей. Пишіть текст для кваліфікованого читача, якому ви пояснюєте, як побудований ваш лексичний аналізатор, що він аналізує саме вашу мову, і що він працює коректно.

6. Базовий та іншу приклади програм вхідною мовою для тестування лексичного аналізатора.

7. План тестування, напр. як у Табл. 3.

8. Протокол тестування у формі текстових копій термінала: фрагмент програми + результат обробки + ваш коментар та/чи оцінка результату

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

План тестування може бути представлений у формі таблиці 3:
Фрагмент протоколу тестування може виглядати, як показано далі.

Тест № 4.2. Програма, всі лексеми якої допустимі у мові `Microlang1`, проте їх використання суперечить синтаксису мови. У цьому випадку лексичний аналізатор помилок НЕ ЗНАХОДИТЬ.
Синтаксично помилкові фрагменти такі:

1. програма починається з лексеми `program1`, а не з `program`;

2. у рядку 3 некоректне використання двох лексем `:=`

№	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	наявність недопустимих лексем	повідомлення про помилку + діагностика, таблиці ідентифікаторів, констант, міток	...
4	надлишок чи відсутність синтаксично виправданих лексем, у т.ч. ключових слів	...	...
5	помилково написані ключові слова	...	...
6	...	...	

Табл. 3. План тестування

```
1 program1 program
2
3 v1 :=:=5.4
4
5 s:=1234
6
7 end
```

Після розбору лексичний аналізатор повідомляє:

```
>python my_lang_lex_example2.py
1   program1   ident      1
1   program    keyword
3   v1         ident      2
3   :=         assign_op
3   :=         assign_op
3   5.4        float      1
5   s          ident      3
5   :=         assign_op
5   1234       int        2
7   end        keyword
Lexer: Лексичний аналіз завершено успішно
```

```
------------------------------
tableOfSymb:{1: (1, 'program1', 'ident', 1),
             2: (1, 'program', 'keyword', ''),
             3: (3, 'v1', 'ident', 2),
             4: (3, ':=', 'assign_op', ''),
             5: (3, ':=', 'assign_op', ''),
             6: (3, '5.4', 'float', 1),
             7: (5, 's', 'ident', 3),
             8: (5, ':=', 'assign_op', ''),
             9: (5, '1234', 'int', 2),
            10: (7, 'end', 'keyword', '')}
tableOfId:{'program1': 1, 'v1': 2, 's': 3}
tableOfConst:{'5.4': 1, '1234': 2}
```

Коментар до тесту № 4.2: лексичний аналізатор правильно розпізнав усі лексеми та визначив їх токени, зокрема `program1` розпізнано як ідентифікатор. Синтаксичну правильність він не перевіряє, тому з точки зору лексики наведена програма коректна і лексер повідомив про успішність лексичного розбору.

Література

[1] Медведєва В.М. Транслятори: лексичний та синтаксичний аналізатори [Текст]: навч.посіб. / В.М. Медведєва, В.А. Третяк. – К.: НТУУ

[2] Ахо, А. Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. / Альфред Ахо, Моника Лам, Рави Сети, Джефри Д. Ульман. – М. : ООО „И.Д. Вильямс”, 2008. – 1184 с.